

OBSTACLE AVOIDANCE PATH
PLANNING BY THE EXTENDED
VGRAPH ALGORITHM

NAGW-1333

By:

C.H. Chung
G.N. Saridis

Department of Electrical, Computer and Systems Engineering
Department of Mechanical Engineering, Aeronautical
Engineering & Mechanics
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

CIRSSE Document #12

OBSTACLE AVOIDANCE PATH PLANNING
BY THE EXTENDED VGRAPH ALGORITHM

by

C. H. Chung and G. N. Saridis

Robotics and Automation Laboratory
Department of Electrical, Computer and Systems Engineering
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

January 1989

CONTENTS

	Page
LIST OF TABLES	iii
LIST OF FIGURES	iv
ABSTRACT	v
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Literature Review	3
2. METHODOLOGY	6
2.1 The Interference Detection	6
2.2 The Grown Space Obstacles	8
2.3 The Rotational Grown Space Obstacles	13
2.4 The VGraph Algorithm	26
2.5 The Graph Search Algorithm	31
2.6 The Orthogonal Projection Method	35
2.7 The Recursive Compensation Algorithm	42
3. PROBLEM STATEMENT, PRELIMINARY RESULTS AND PROPOSED WORK	56
3.1 Problem Statement	56
3.2 Preliminary Results	57
3.3 Proposed Work	60
REFERENCES	61
Appendix A: Simulation of the VGraph Algorithm	66
Appendix B: I/O Files for the VGraph Algorithm	101
Appendix C: Simulation of the Rotational GSpace	108
Appendix D: I/O Files for the Rotational GSpace	116
Appendix E: Simulation of the Branch and Bound Algorithm ..	120
Appendix F: I/O Files for the Branch and Bound Algorithm ..	126
Appendix G: Simulation of the RCA	127
Appendix H: I/O Files for the RCA	134
Appendix I: Simulation of the OPM	137
Appendix J: I/O Files for the OPM	148

LIST OF TABLES

	Page
Table 2.4.1 Simulation result of the VGraph Algorithm	28
Table 2.7.1 Euclidean distance and Computing time	45
Table 3.2.1 Simulation result of the VGraph Algorithm	58
Table 3.2.4 Euclidean distance and Computing time	60

LIST OF FIGURES

	Page
Fig. 2.1.1 Interference Detection	6
Fig. 2.1.2 Dead Node	7
Fig. 2.2.1 A description of Workspace A	9
Fig. 2.2.2 A description of GSpace Obstacles for Workspace A	10
Fig. 2.2.3 The Grown Space Obstacles for Workspace A	11
Fig. 2.2.4 Data structure for Workspace A	12
Fig. 2.3.1 A description of Workspace B	15
Fig. 2.3.2 A description of the rotational GSpace Obstacles	16
Fig. 2.3.3 The rotational Grown Space Obstacles	17
Fig. 2.3.4 Data structure of the rotational GSpace Obstacles	18
Fig. 2.3.5 Workspace C for the Problem Statement 2.3	19
Fig. 2.3.6 The rotational GSpace Obstacles with 0 sliced	20
Fig. 2.3.7 The rotational GSpace Obstacles with $\frac{\pi}{6}$ sliced	21
Fig. 2.3.8 The rotational GSpace Obstacles with $\frac{\pi}{3}$ sliced	22
Fig. 2.3.9 The rotational GSpace Obstacles with $\frac{\pi}{2}$ sliced	23
Fig. 2.3.10 The rotational GSpace Obstacles with $\frac{2\pi}{3}$ sliced	24
Fig. 2.3.11 The rotational GSpace Obstacles with $\frac{5\pi}{6}$ sliced	25
Fig. 2.4.1 A VGraph for Workspace A	29
Fig. 2.4.2 The collision-free shortest path for Workspace A	30
Fig. 2.6.1 A description of Workspace D	37
Fig. 2.6.2 A description of three orthogonal projections	38
Fig. 2.6.3 A description of Grown Space Obstacles in 3D	39
Fig. 2.6.4 A reconstruction of Grown Space Obstacles in 3D	40
Fig. 2.6.5 The Grown Space Obstacles of Workspace D	41
Fig. 2.7.1 The path calculated by the VGraph algorithm	51
Fig. 2.7.2 The first compensation by the RCA	52
Fig. 2.7.3 The second compensation by the RCA	53
Fig. 2.7.4 The final path by the RCA	54
Fig. 2.7.5 The Euclidean distance by the RCA	55

ABSTRACT

In many path planning algorithms, attempts are made to optimize the path between the *start* and the *goal* in terms of *Euclidean* distance. Since the moving object is shrunk to a point in the *Configuration Space*, *Findpath* can be formulated as a graph searching problem. This is known as the *VGraph Algorithm*.

Lozano-Pérez points out the drawbacks of the *VGraph Algorithm*. The first drawback is related with rotation of a moving object. This drawback can be solved by using the *sliced projection method*. However, the *VGraph Algorithm* has serious drawbacks when the obstacles are three-dimensional. The *Extended VGraph Algorithm* is proposed to solve the drawbacks of the *VGraph Algorithm* by using the *Recursive Compensation Algorithm*. The *Recursive Compensation Algorithm* is proposed to find the collision-free shortest path in 3D and it is proved to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph*.

1. INTRODUCTION

1.1 Motivation

In many path planning algorithms, attempts are made to optimize the path between the start and the goal in terms of *Euclidean* distance. In the *Configuration Space* [9], the moving object is shrunk to a *Configuration Point*, while the stationary obstacles are expanded to fill all space where the presence of the *Configuration Point* would imply a collision of the object with obstacles. Therefore, *Findpath* [29] can be formulated as a graph searching problem. The graph is formed by connecting all pairs of visible vertices of the *Configuration Space Obstacles*.

Consider the *VGraph Algorithm* for a moving object to find the collision-free shortest path in a workspace with some obstacles. A lot of work has been done in this field, which has the following design steps:

- Build the *Grown Space Obstacles*.
- Find the visible vertices by detecting interferences.
- Build the *VGraph* with a set of the visible vertices.
- Search the *VGraph* by the graph search algorithm.

The shortest path from the start to the goal in this *VGraph Algorithm* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez [29] points out the drawbacks of the *VGraph Algorithm*. The first drawback is related with the rotation of a moving object. Since the *VGraph Algorithm* require moving an object along obstacle boundaries, shortest paths are very susceptible to inaccuracies in the object models. This drawback can be solved by using the *sliced projection method* [28] [29] [30]. However, the *VGraph Algorithm* has serious drawbacks [29] when the obstacles are three-dimensional:

- shortest paths do not typically traverse the vertices of the *Grown Space Obstacles*,
- there may be no paths via vertices, within the enclosing polyhedral region R , although other types of safe paths within R may exist.

Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the

Grown Space Obstacles. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. So, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D. Therefore, a new algorithm, called the *Extended VGraph Algorithm*, should deal with the drawbacks of the *VGraph Algorithm*.

The *Extended VGraph Algorithm* has the following design steps;

- 1) Apply the *Orthogonal Projection Method* to get the *Grown Space Obstacles* in 3D.
 - i) Project obstacles in 3D onto the projection spaces.
 - ii) Build the *Grown Space Obstacles* in 2D.
 - iii) Select the necessary *Grown Space Obstacles* for the *VGraph*.
 - iv) Reconstruct the *Grown Space Obstacles* in 3D.
- 2) Find the visible vertices by detecting interferences.
- 3) Build the *VGraph* with a set of the visible vertices.
- 4) Search the *VGraph* by the graph search algorithm.
- 5) Apply the *Recursive Compensation Algorithm* to obtain the collision-free shortest path in 3D.

The following results have been obtained by the *Extended VGraph Algorithm*;

- The *Extended VGraph Algorithm* can deal with not only translations of a moving object but also its rotations by using the θ sliced projection method.
- Since the *Orthogonal Projection Method* avoids building the unnecessary *Grown Space Obstacles*, it can make the *VGraph* simpler than any other algorithms that use all of the *Grown Space Obstacles*. Therefore, the *Orthogonal Projection Method* can save the memory space to store the representation of the *Grown Space Obstacles* and it can shorten the graph search time because of the simpler *VGraph*.

- The *Recursive Compensation Algorithm* can guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph*. The property of convergency of the *Recursive Compensation Algorithm* is proved. Since ϵ is set to 10^{-5} , Lozano-Pérez's alleviation method needs a lot of memory space to store $(2 + 8 \times n \times \epsilon^{-1})$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2 + 8 \times n)$ vertices for the *VGraph*. The accuracy is defined by ϵ whose value is very small and n is the number of obstacles in workspace. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time.
- The *Extended VGraph Algorithm* has been presented to solve the drawbacks of the *VGraph Algorithm*.

1.2 Literature Review

The simplest obstacle avoidance algorithm uses the *Generate and Test Method* [16] [28]. A simple path from *Start* to *Goal* is hypothesized and is tested for potential collisions. If a collision is expected, a new path is considered. This process is repeated until no collisions are expected along the new proposed path. In the case of a manipulator, such an algorithm can be described in three steps:

- 1) Calculate the volume swept out by the manipulator along the proposed path.
- 2) Determine the overlap between obstacles and the swept volume by a manipulator.
- 3) Propose a new path.

The first step is self explanatory. The second step is known as an *Interference Detection* [7], detecting the overlap between the obstacles and the swept volume by manipulator. The whole 3 steps are known as the *Swept Volume Method* [28]. Lozano-Pérez [28] and Faverjon [16] have pointed out several difficulties and drawbacks of the *Swept Volume Method*. First, it is quite difficult to model obstacles and a manipulator within resonably short computational time and allowable accuracy. Calculating the volume swept out by a manipulator with revolute joints is a hard job. It can be also difficult to determine whether the swept volume and obstacles overlap. Another fundamental drawback lies in the relationship between the second and the third steps. Each proposed path provides only local information about potential collisions, for example, the shape of the intersections of the volumes involved, or the identity of the obstacle giving rise to the collision. As the manipulator

consists of several linked parts, it is difficult to find good heuristics to modify the paths. This lack of a global view can result in an expensive search of the space of possible paths with a very large upper bound on the worst case length of the path. For these reasons, Udupa [59] uses a *Growing Transformation Method* on obstacles to compute approximations to the forbidden regions for the three-dimensional reference point of a three degree of freedom subset of a manipulator [28]. The system maintains a variable resolution description of the legal positions of the reference point. Safe paths for the subset manipulator are found by recursively introducing intermediate goals into a straight line path until the complete path is in free space. This method has two drawbacks pointed by Lozano-Pérez [28] [30].

- 1) Since the complete manipulator has more than three degrees of freedom, the three-dimensional forbidden regions cannot model all the constraints on the manipulator. When a trajectory fails, Udupa's system makes a correction using manipulator dependent heuristics. The use of heuristics tends to limit the performance of the algorithm in cluttered spaces.
- 2) The recursive path finder uses only local information to determine a safe path and therefore suffers from some of the same drawbacks as the *Swept Volume Method*.

Lozano-Pérez [28] generalized the ideas of Udupa [59] to the whole manipulator. His algorithm uses a more accurate growing operation to compute the forbidden regions in both two and three-dimensions. It introduces a graph searching technique for path finding, which produces optimum two-dimensional paths when only translations are involved. The algorithm is then generalized to deal with three-dimensional obstacles and extended to deal uniformly with more than three degrees of freedom. However, the generalization to three-dimensions has an unfortunate side effect. The shortest path around a polyhedral obstacle does not in general traverse only vertices of the polyhedron. That is, the shortest path in the *VGraph* whose node set contains only vertices of the grown obstacles is not guaranteed to be the shortest collision-free path. So, Lozano-Pérez [28] proposed a method which is to introduce additional vertices along the edges of the grown obstacles so that no edge is longer than a prespecified maximum length. This approach has some drawbacks. It is difficult to decide how many vertices should be added along the edges of the grown obstacles and the additional vertices need much more computational time for the *VGraph* search.

Brooks [8] solves the *Findpath* problem by good representation of free space; Ahuja [4] and Faverjon [16] use an *Octree* for the obstacle avoidance. Brooks [10] presents an algorithm for polyhedral obstacles and a moving object with two translational and one rotational degrees of freedom. Wong and Fu [63] present a methodology for three-dimensional collision-free path planning by which planning is done

in the three-dimensional orthogonal two-dimensional projections of a three-dimensional environment. Peshkin and Sanderson [53] present an algorithm that efficiently finds the externally visible vertices of a polygon and the range of angles. Chung and Saridis [11] present the *Recursive Compensation Algorithm* to solve the drawback of the *VGraph Algorithm*.

2. METHODOLOGY

2.1 The Interference Detection

Boyse [7] presents two types of interference checking: detection of intersections among objects in fixed positions and detection of collisions among objects moving along specified trajectories. The first type of interference checking plays an important role in the *Interference Detection* of the Grown Space Obstacles and the second type of interference checking plays an important role in *Obstacle Avoidance*. To detect a collision between two objects, it is sufficient [7] to detect a collision of an edge on one object with a face of the other or vice-versa. Because a face consists of its interior and a boundary, collision of a face and edge occurs in one of two ways; the edge comes into contact either with the interior of the face or with the boundary of the face. The two cases are shown in Fig. 2.1.1 [7].

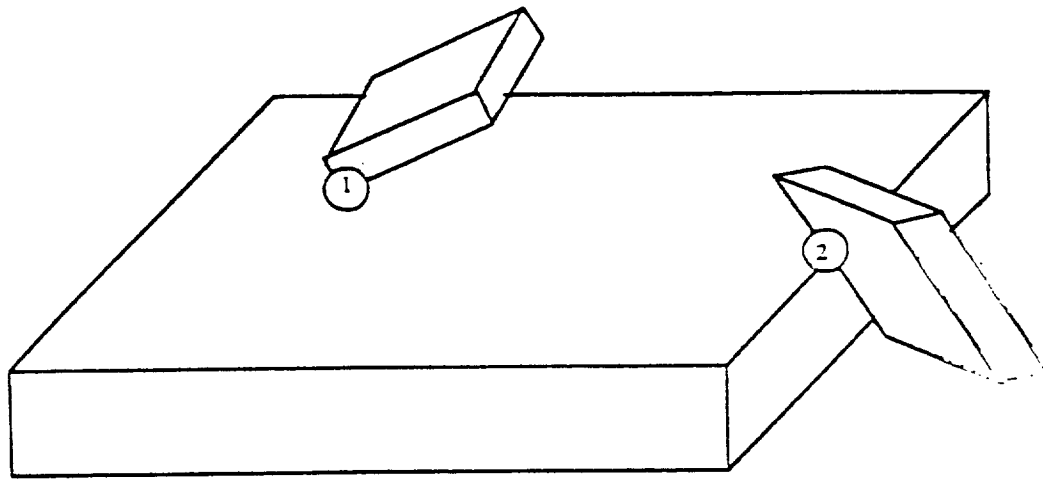


Fig. 2.1.1 Interference Detection

The collision detection algorithm [7] considers each of these two possible situations as follows:

1. *Edge contacts face interior.* Because edges are straight line segments and faces are planar, contact must occur at an endpoint of the edge. Assuming an edge moving relative to a stationary face, collision can be detected by determining the locus of each endpoint of the moving edge and examining these loci (space curves) to see whether either one intersects the face.
2. *Edge contacts face boundary.* Again assume an edge moving relative to a stationary face and note that the locus of this moving edge generates a surface in space. Collision is detected by examining the

boundary of the face to see if it intersects the surface generated by the moving edge.

The first type of interference will be detected by the *Dead Node*, defined as a vertex that is located in the object, shown in Fig. 2.1.2. To be a *Dead Node*, its boundary condition and its boundary equation should be satisfied.

(i) boundary condition

$$x_{min} < P_x < x_{max}$$

$$y_{min} < P_y < y_{max}$$

(ii) boundary equation

$$f_1(x, y) \cdot f_2(x, y) \cdot f_3(x, y) \cdot f_4(x, y) > 0$$

The second type of interference can be detected by checking the *Line Intersection* [57]. The straight forward way [57] to solve this problem is to find the intersection point of the lines defined by the line segments, then check whether this intersection point falls between the end points of both of the segments. In terms of the variables in Sedgewick's algorithm [57], it is easy to check that the quantity $(dx \cdot dy_1 - dy \cdot dx_1)$ is 0 if p_1 is on the line, positive if p_1 is on one side, and negative if it is on the other side. The same holds true for the other point, so the product of the quantities for the two points is positive if and only if the points falls on the same side of the line, negative if and only if the points fall on different sides of the line, and 0 if and only if one or both points fall on the line.

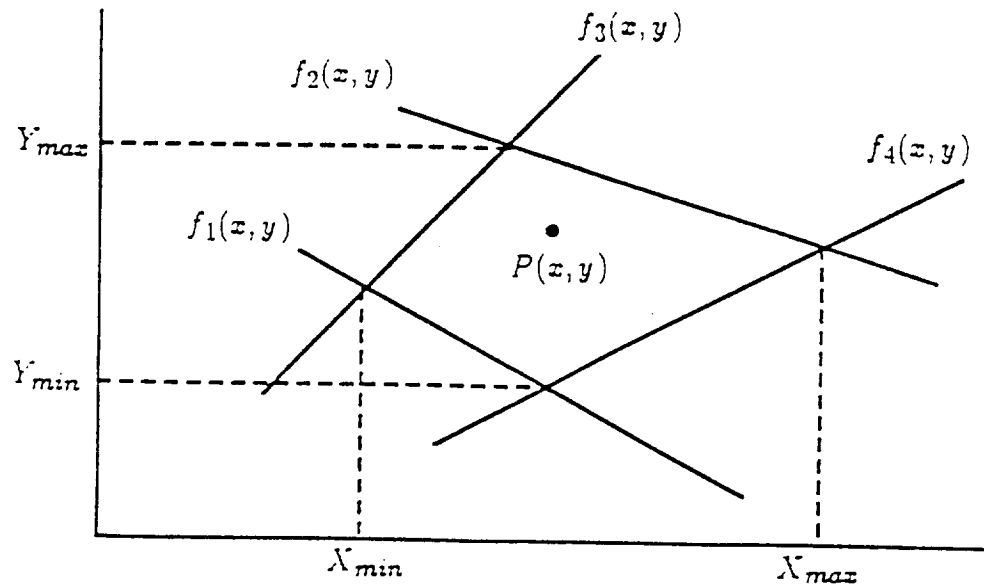


Fig. 2.1.2 Dead Node

2.2 The Grown Space Obstacles

Most of the work in path planning will be done in the field of building the *Configuration Space Obstacles* rather than searching graph. Therefore, it is clear that the representation of the objects [4] plays a major role in determining the feasibility and performance of any intersection or collision detection method using that representation. Udupa [58] was the first to approach the *Findpath* by explicitly using transformed obstacles and a space where the moving object is shrunk to be a point. Udupa used only rough approximation to the actual *Configuration Space Obstacles* and had no direct method for representing constraints on more than three degrees of freedom [30]. Lozano-Pérez [29] [30] shows that algorithms for computing the *Grown Space Obstacles* in 2D have time complexity $O(v)$, and the algorithms for computing the *Grown Space Obstacles* in 3D have time complexity $O(v^2 \log v)$, where v is the total number of vertices. Considering time complexity, it is much better to find a collision-free path projected in 2D rather than in 3D. Therefore, the *Orthogonal Projection Method* is proposed to build the *Configuration Space Obstacles* in 3D, where three orthogonal cameras are used to build the *Configuration Space Obstacles*. To avoid building the *Grown Space Obstacles* of unnecessary objects in 3D has a lot of advantages. See the section 2.6 for these advantages. A final *Configuration Space Obstacles* in 3D will be reconstructed from the three *Configuration Space Obstacles* in 2D.

Workspace A (Fig. 2.2.1, Fig. 2.2.2 and Fig. 2.2.3) demonstrates Udupa's idea to build the *Grown Space Obstacles* in 2D. Fig. 2.2.1 describes *Workspace A* with three obstacles and the initial and goal states of a moving object. Fig. 2.2.2 describes how to build the *Grown Space Obstacles* with respect to a reference point. The moving object is applied to the boundary of each object and the reference point is traced to obtain the *Grown Space Obstacles*. So, the moving object is shrunk to be a point and the grown geometric objects are obtained, called *Grown Space Obstacles*, that represent all the positions of the moving object that cause collision with the obstacles. Fig. 2.2.3 describes the final *Grown Space Obstacles* for *Workspace A*. The advantage of this formulation [30] is that the intersection of a point relative to a set of objects is easier to deal with than the intersection of objects among themselves. Fig. 2.2.4 describes the data structure for *Workspace A*. Representing the positions of rigid objects requires specifying all their degrees of freedom, both translations and rotations. The configuration [30] of a polyhedron is a set of independent parameters that characterize the position of every point in the object. In following sections, the different initial configuration of a moving object makes the different *Grown Space Obstacles*, which result in different *VGraph*.

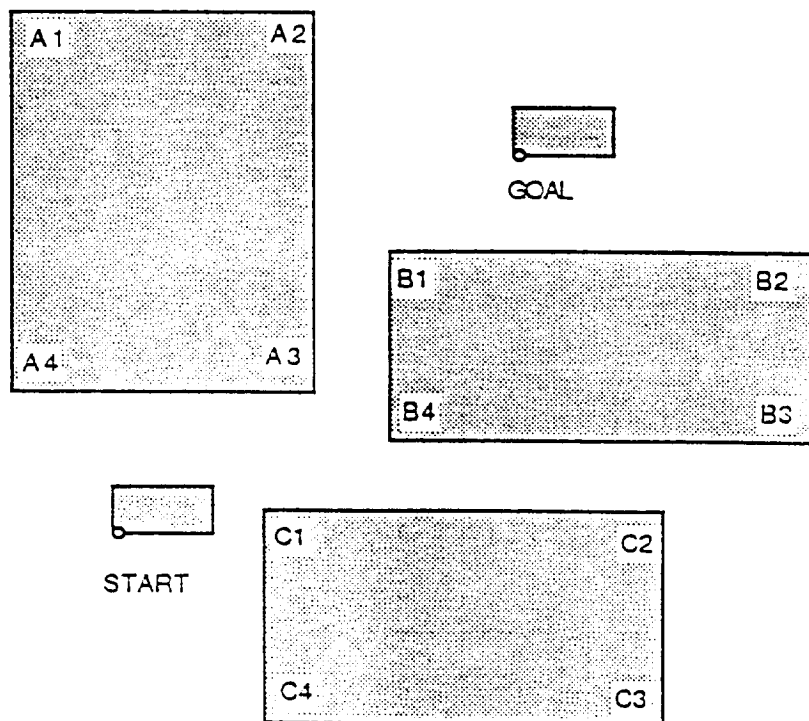


Fig. 2.2.1 A description of Workspace A

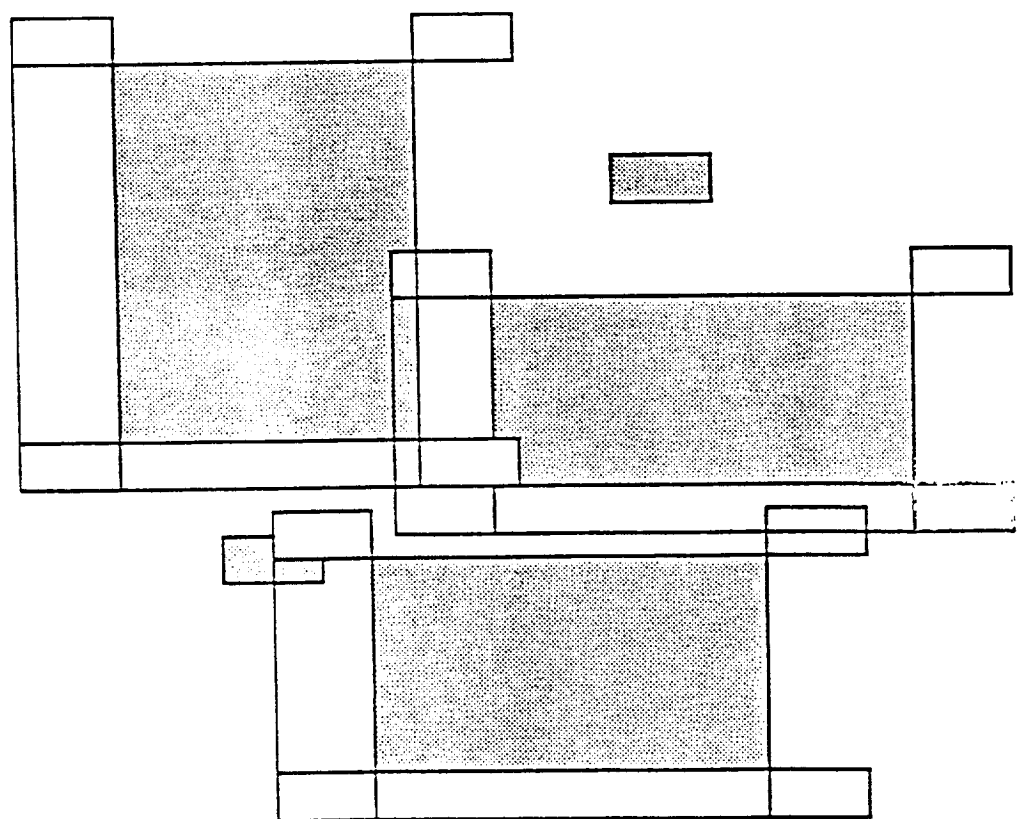


Fig. 2.2.2 A description of Grown Space Obstacles for Workspace A

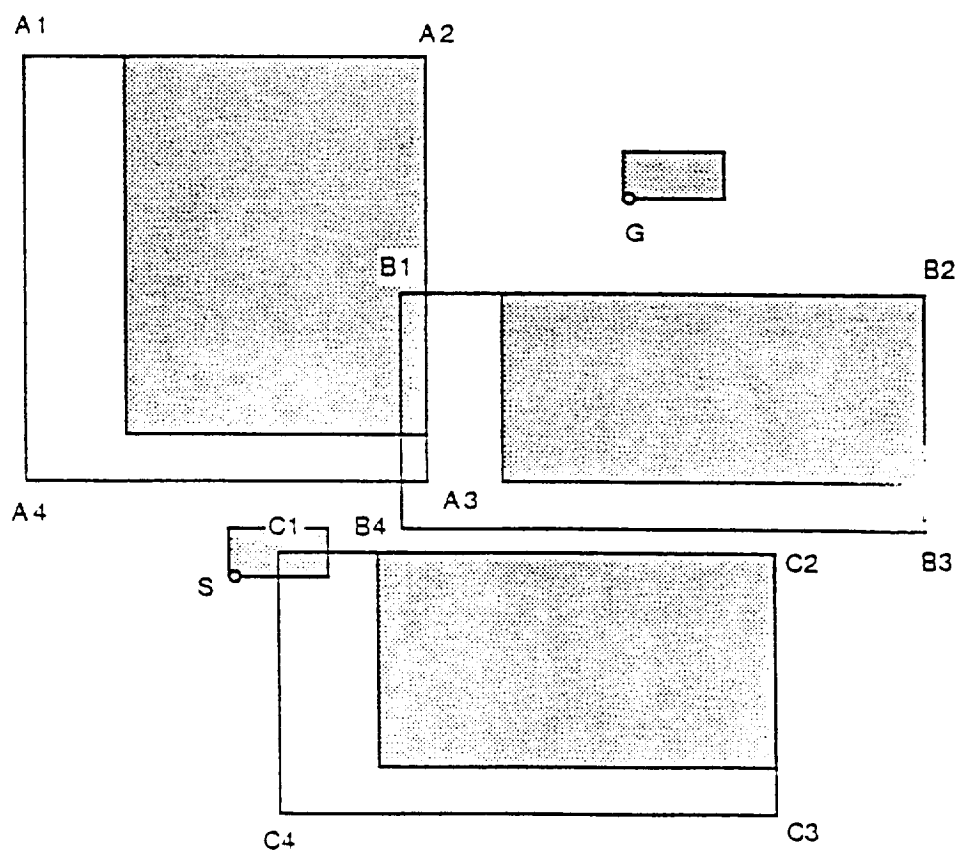


Fig. 2.2.3 A Grown Space Obstacles for Workspace A

Objects (linked list)

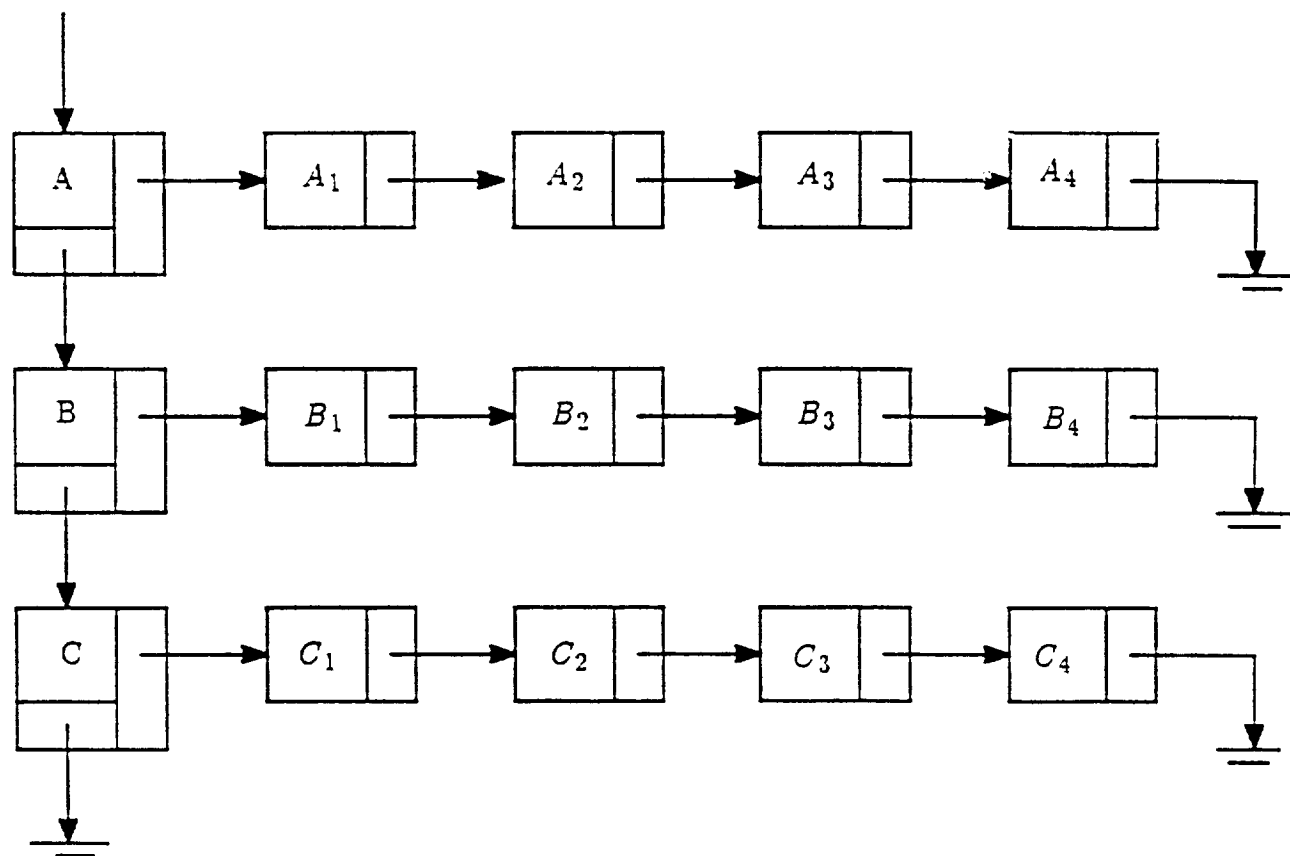


Fig. 2.2.4 Data structure for Workspace A

2.3 The Rotational Grown Space Obstacles

In the previous section, it is known that the different initial configuration of a moving object makes the different *GSpace Obstacles* (*Grown Space Obstacles*), which result in different *VGraphs*. Since the rotation of a moving object can change its initial configuration, each *VGraph* should be built for its rotation of the moving object. Lozano-Pérez [29] [30] presents the θ sliced projection method to build the *GSpace Obstacles* for the rotation of the moving object. Since the moving object can rotate by θ , the number of its *VGraph*, ξ , can be figured out, where $0 \leq \eta \cdot \theta \leq 2\pi$ and $\eta = 1, 2, \dots, \xi$. For each η , build its *VGraph* and construct a set of vertices for all vertices for its *VGraph*. Find the visible vertices from this set of vertices by detecting the interferences. This set of visible vertices can build the *VGraph* with the sliced rotation of a moving object.

Let's consider Workspace B, shown in Fig. 2.3.1, having a moving object rotated by θ with respect to Workspace A. Fig. 2.3.2 describes how to build the *Grown Space Obstacles* with respect to a reference point. The moving object is applied to the boundary of each object and the reference point is traced to obtain the rotational *Grown Space Obstacles*. So, the moving object is shrunk to be a point and the grown geometric objects are obtained, representing all the positions of the moving object that cause collision with the obstacles. Fig. 2.3.3 describes the *Grown Space Obstacles* for Workspace B. Workspace A and Workspace B have the same configuration except the initial configuration of the moving object. However, they have the completely different *Grown Space Obstacles*, shown in Fig. 2.2.3 and Fig. 2.3.3. Fig. 2.3.4 describes a data structure of the rotational *GSpace Obstacles*. The vertices of the rotational *GSpace Obstacles*, shown in Fig. 2.3.3, can be obtained from the geometric equations, assuming that h is the horizontal length of a moving object, v is its vertical length, θ is the radian angle between the initial configuration and the rotated configuration with respect to the reference point. And the lower character means the vertices of the obstacles and the upper character means the vertices of the *GSpace Obstacles*, i.e., $a_i = (a_{ix}, a_{iy})$, $A_i = (A_{ix}, A_{iy})$ where $i = 1, \dots, n$. However, if θ is 0 or $\frac{\pi}{2}$, then the set of A_{odd} equals to the set of A_{even} . Fig. 2.3.4 shows how to design the data structure to store the information on the rotational *Grown Space Obstacles* to save the memory storage.

where $0 \leq \theta < \frac{\pi}{2}$

$$A_1 = (a_{1x}, a_{1y}) + h \cdot (-\cos\theta, -\sin\theta)$$

$$A_2 = (a_{1x}, a_{1y})$$

$$A_3 = (a_{2x}, a_{2y})$$

$$A_4 = (a_{2x}, a_{2y}) + v \cdot (\sin\theta, -\cos\theta)$$

$$A_5 = (a_{3x}, a_{3y}) + v \cdot (\sin\theta, -\cos\theta)$$

$$A_6 = (A_{5x}, A_{5y}) + h \cdot (-\cos\theta, -\sin\theta)$$

$$A_7 = (A_{6x}, A_{6y}) + v \cdot (\sin\theta, -\cos\theta)$$

$$A_8 = (a_{4x}, a_{4y}) + h \cdot (-\cos\theta, -\sin\theta).$$

where $\frac{\pi}{2} \leq \theta < \pi$

$$\theta \leftarrow \theta - \frac{\pi}{2}$$

$$h \leftarrow v$$

$$v \leftarrow h.$$

Lozano-Pérez points out two important properties of sliced projection:

- i) a solution to a Findspace problem in any in the slices is a solution to the original problem, but since the slices are an approximation to the *Grown Space Obstacles*, the converse is not necessarily true;
- ii) the slice projection of a *Grown Space Obstacles* can be computed by using the swept volume operation, without having to compute the high-dimensional *Grown Space Obstacles*.

When rotations of a moving object are allowed, the slice projection operation can be used to extend the *VGraph Algorithm* to find safe paths [29].

[Problem Statement 2.3] Assuming that the horizontal length of the moving object is 2, its vertical length is 1 and θ is $\frac{\pi}{6}$ and obstacles are given as in Fig. 2.3.5, draw the rotational *GSpace Obstacles*.

Fig. 2.3.6 - Fig. 2.3.11 draw the rotational *GSpace Obstacles*. The programming list for the simulation of the rotational *GSpace Obstacles* is available in Appendix C and Appendix D.

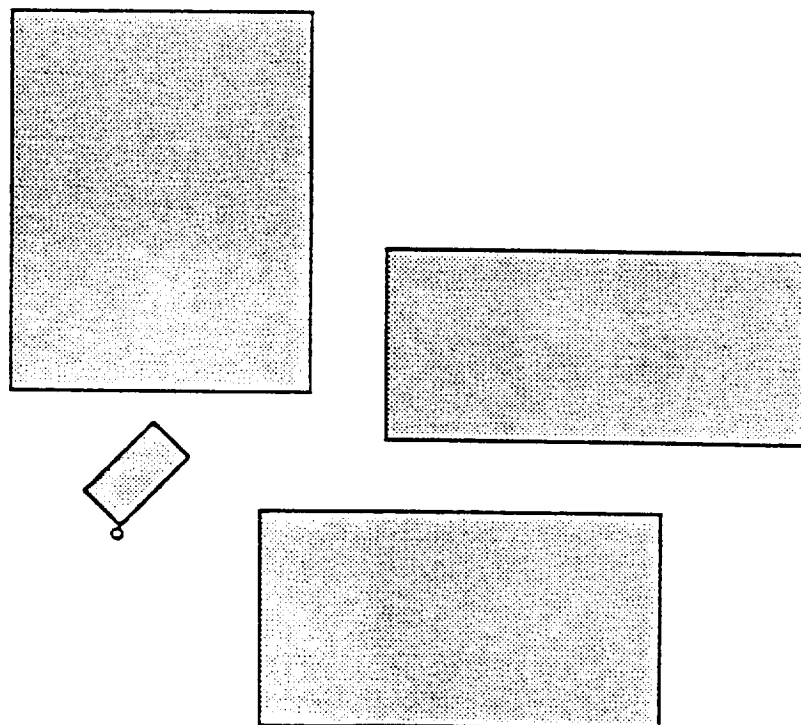


Fig. 2.3.1 A description of Workspace B

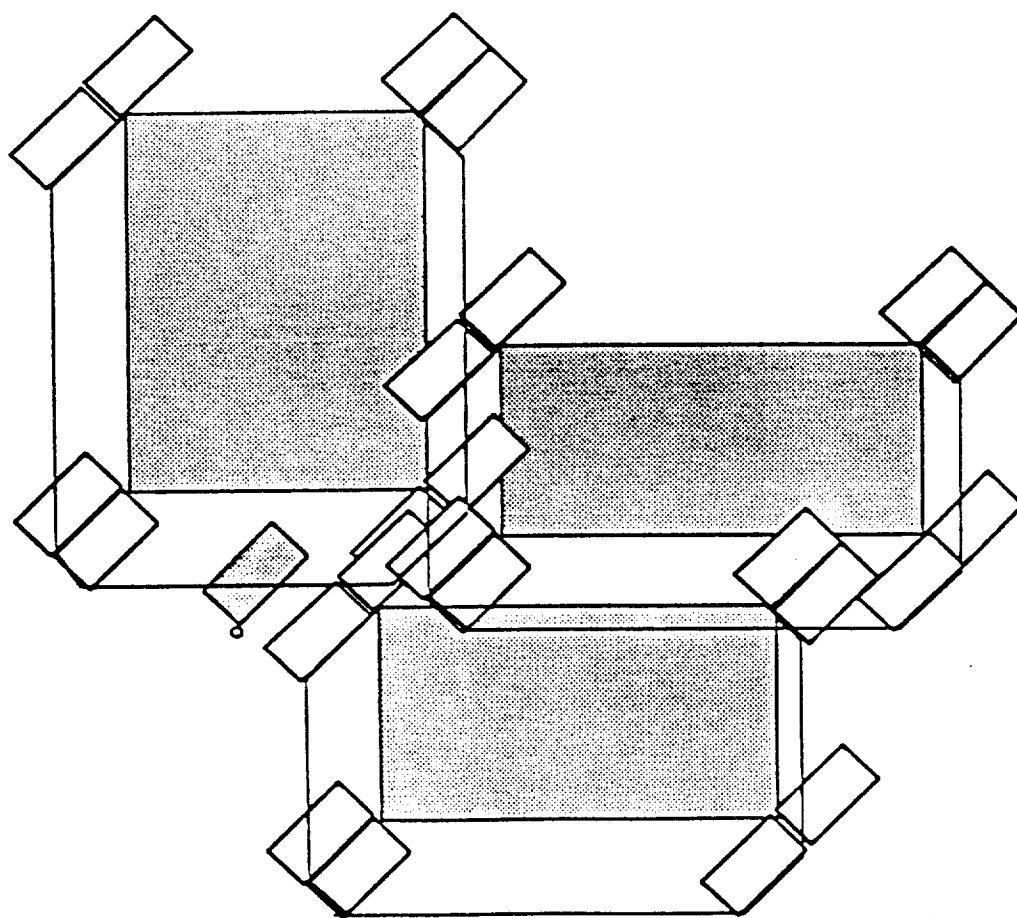


Fig. 2.3.2 A description of the rotational GSpace

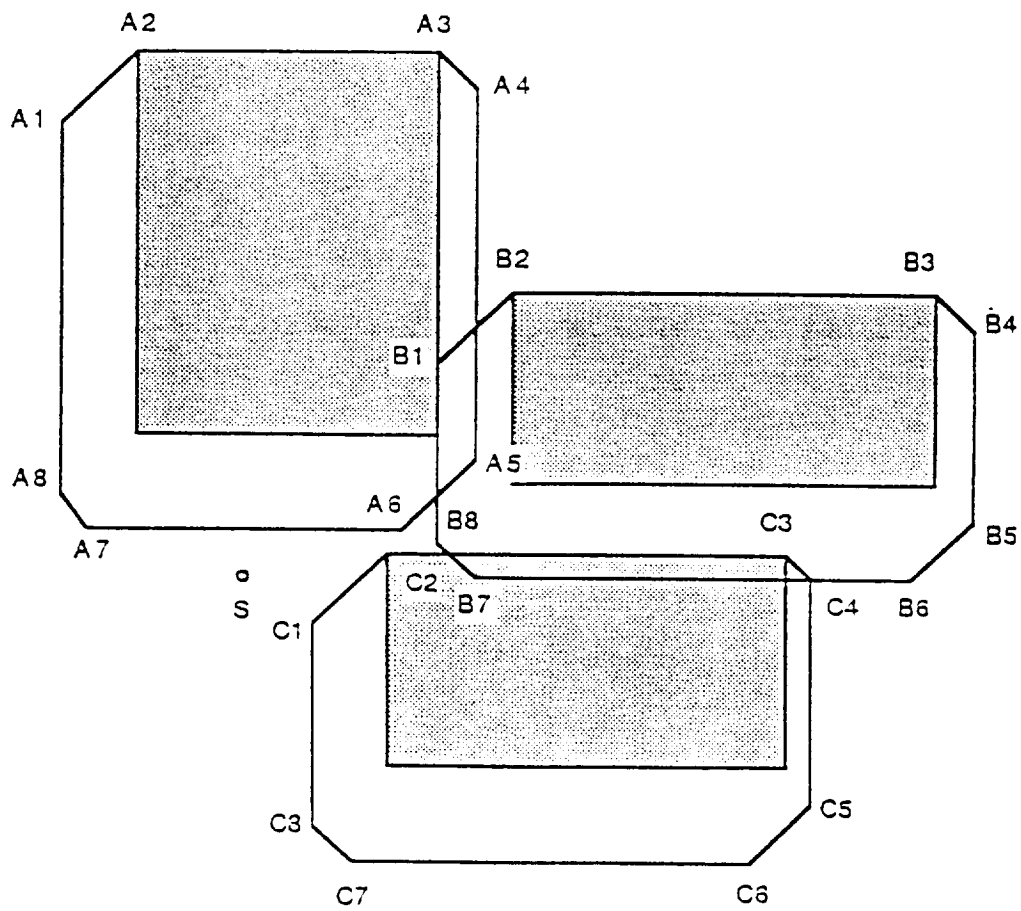


Fig. 2.3.3 A rotational GSpace

Object A (linked list)

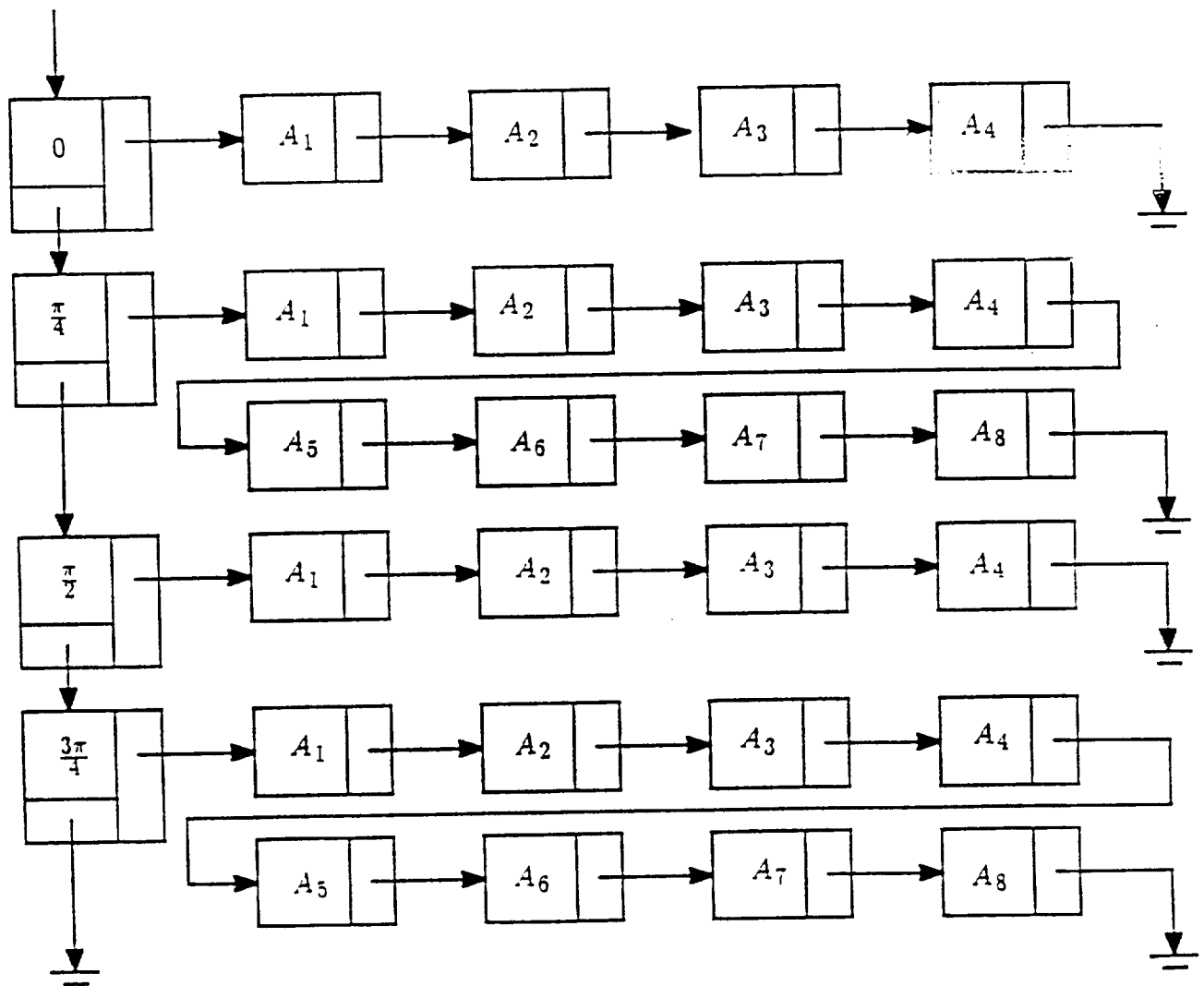


Fig. 2.3.4 Data structure of the rotational GSpace

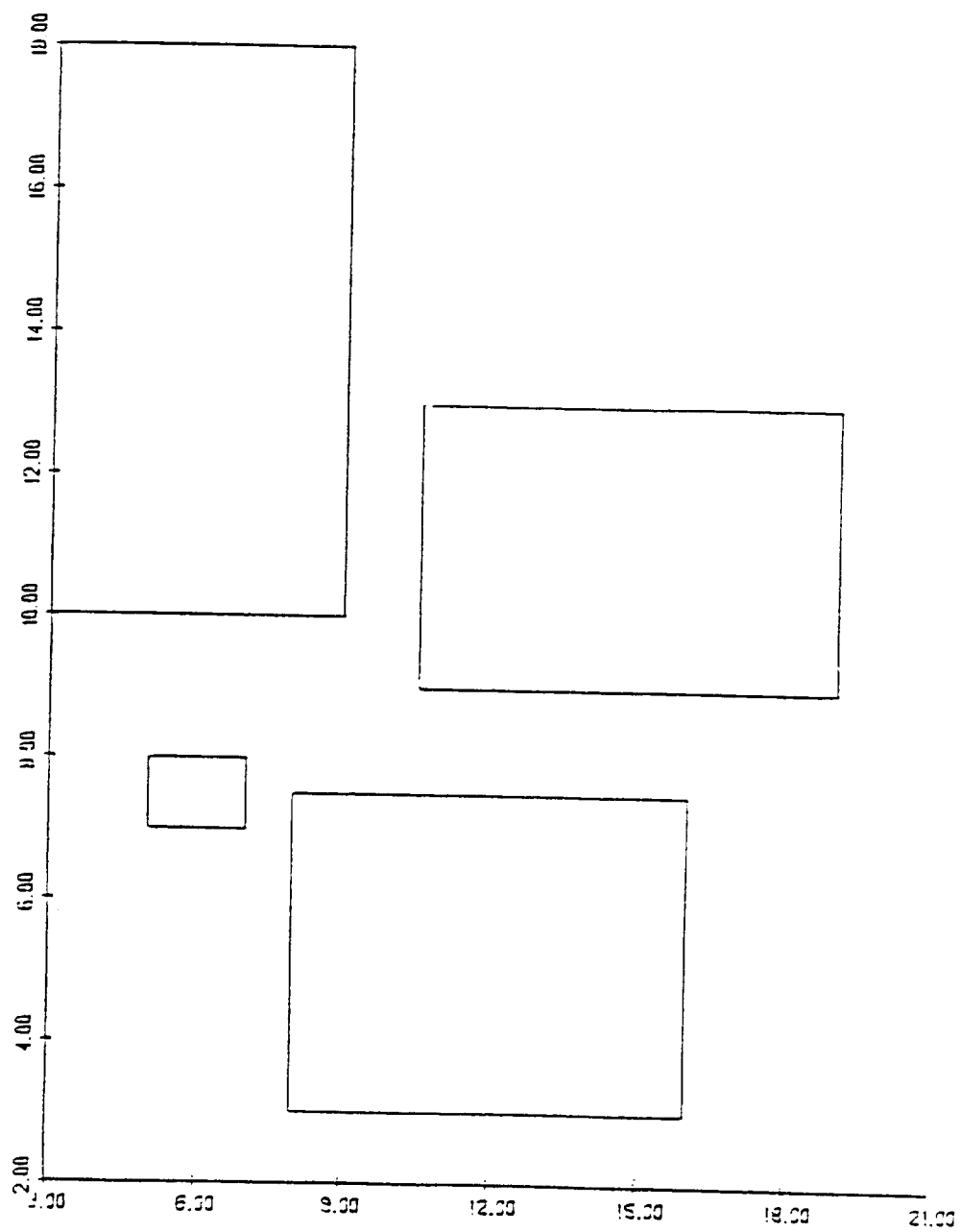


Fig. 2.3.5 Workspace C for the Problem Statement 2.3

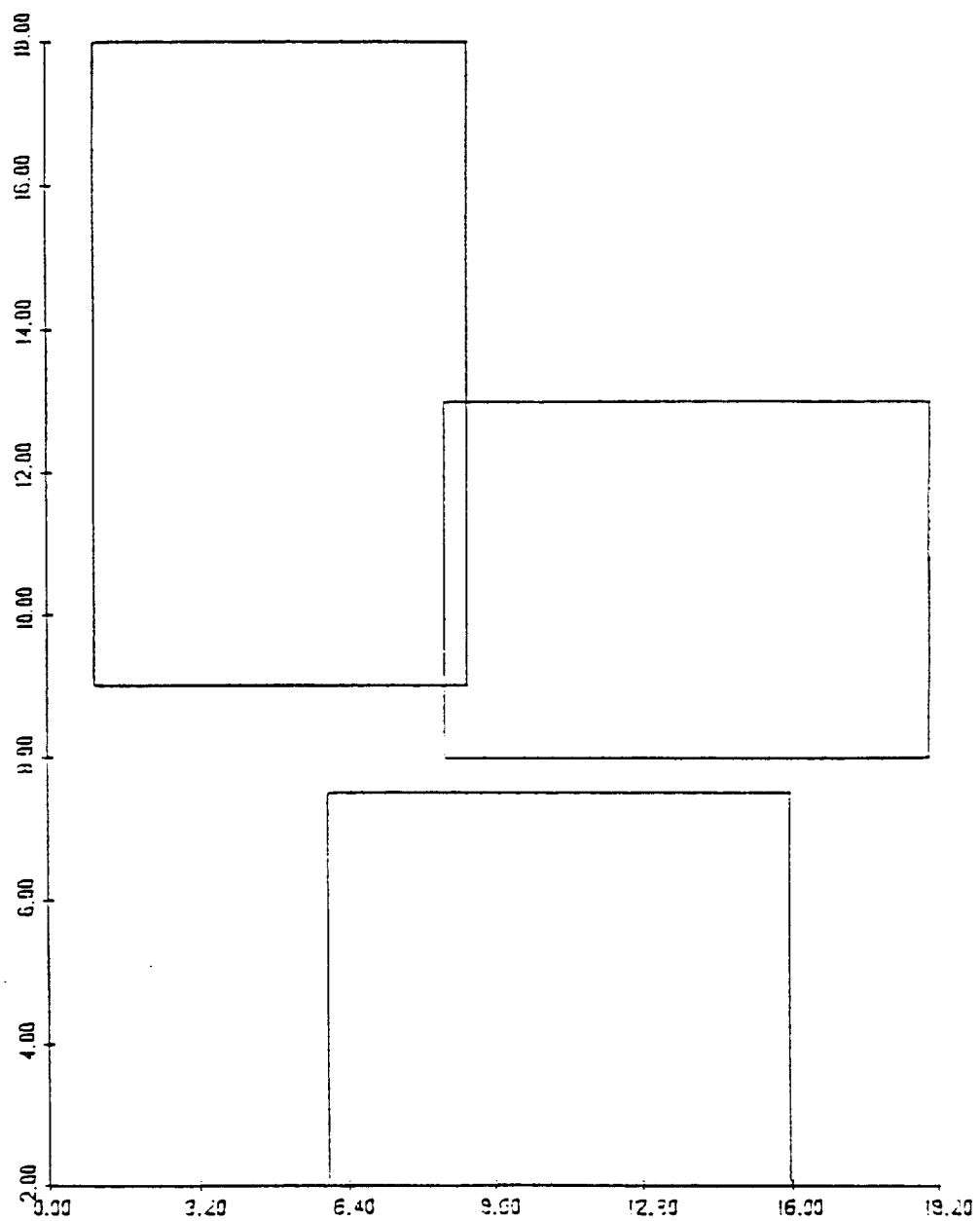


Fig. 2.3.6 A rotational GSpace with 0 sliced

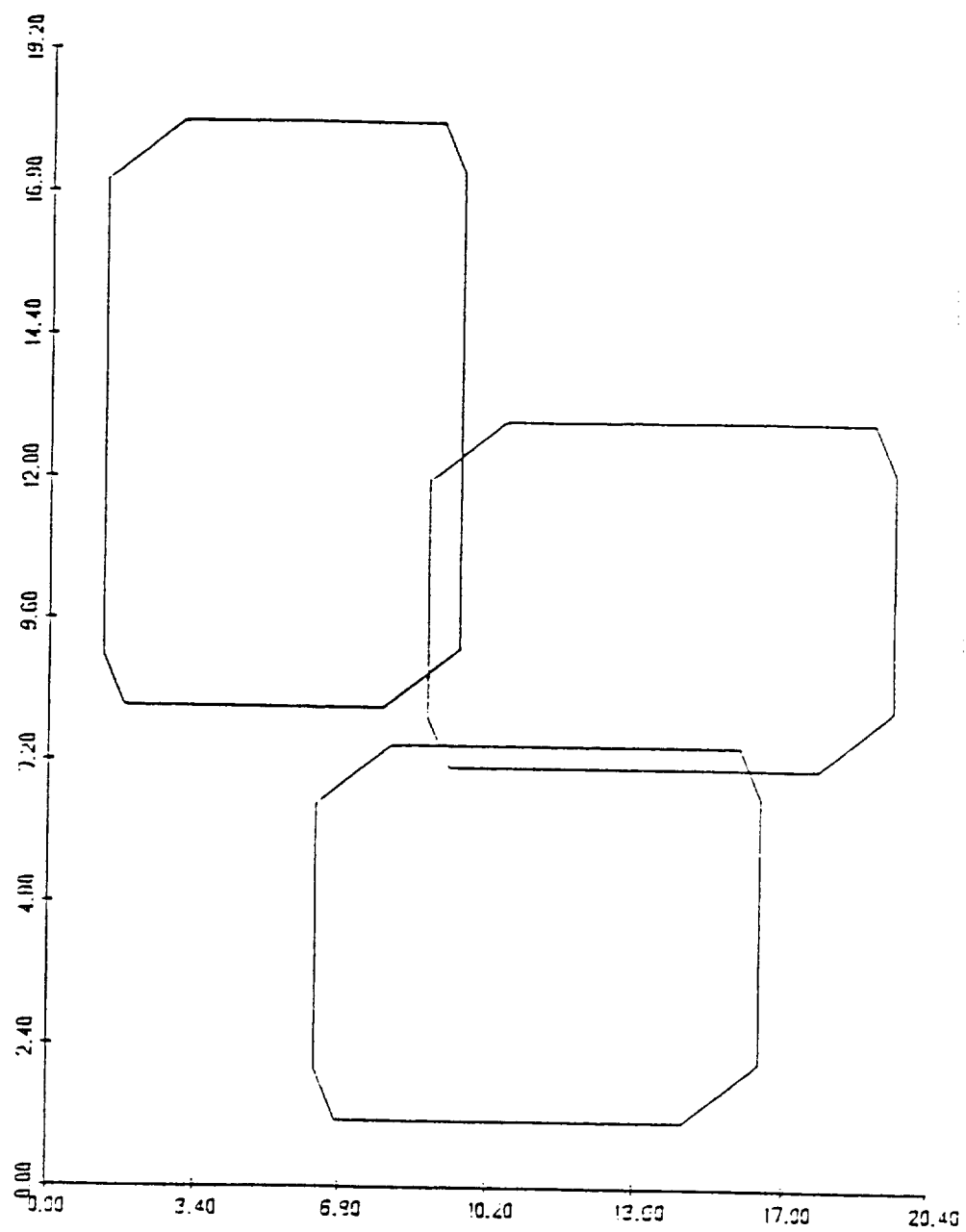


Fig. 2.3.7 A rotational GSpace with $\frac{\pi}{6}$ sliced

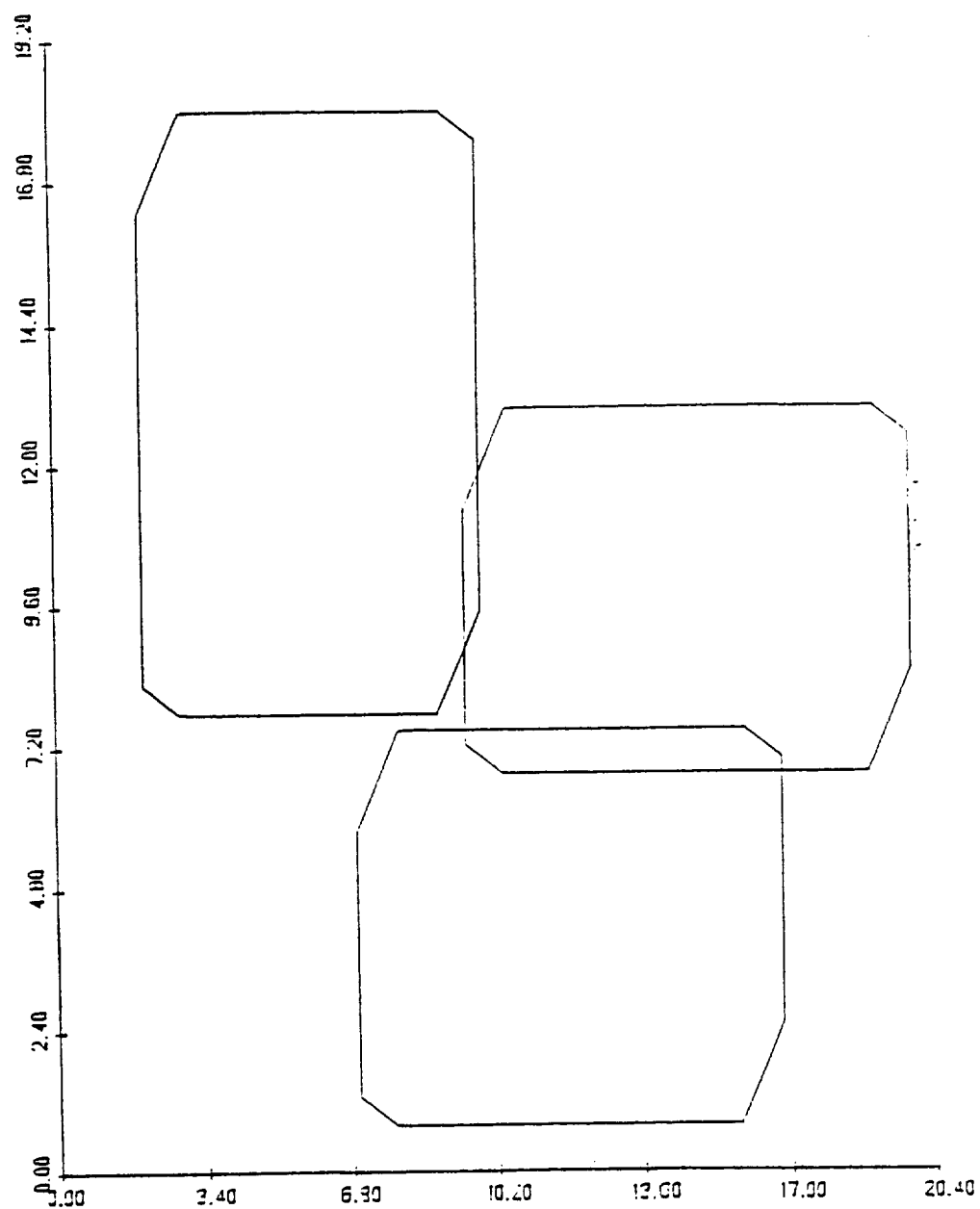


Fig. 2.3.8 A rotational GSpace with $\frac{\pi}{3}$ sliced

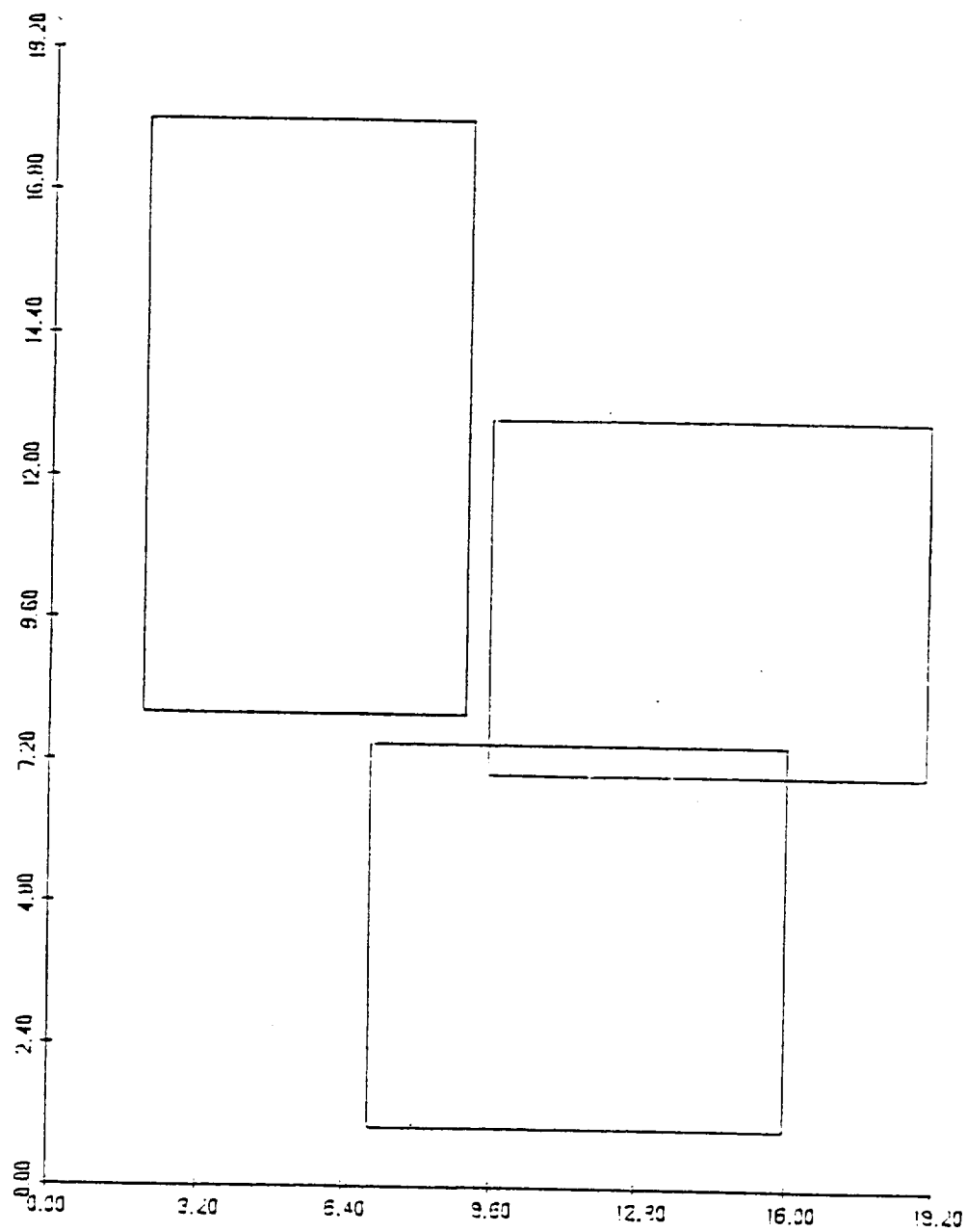


Fig. 2.3.9 A rotational GSpace with $\frac{\pi}{2}$ sliced

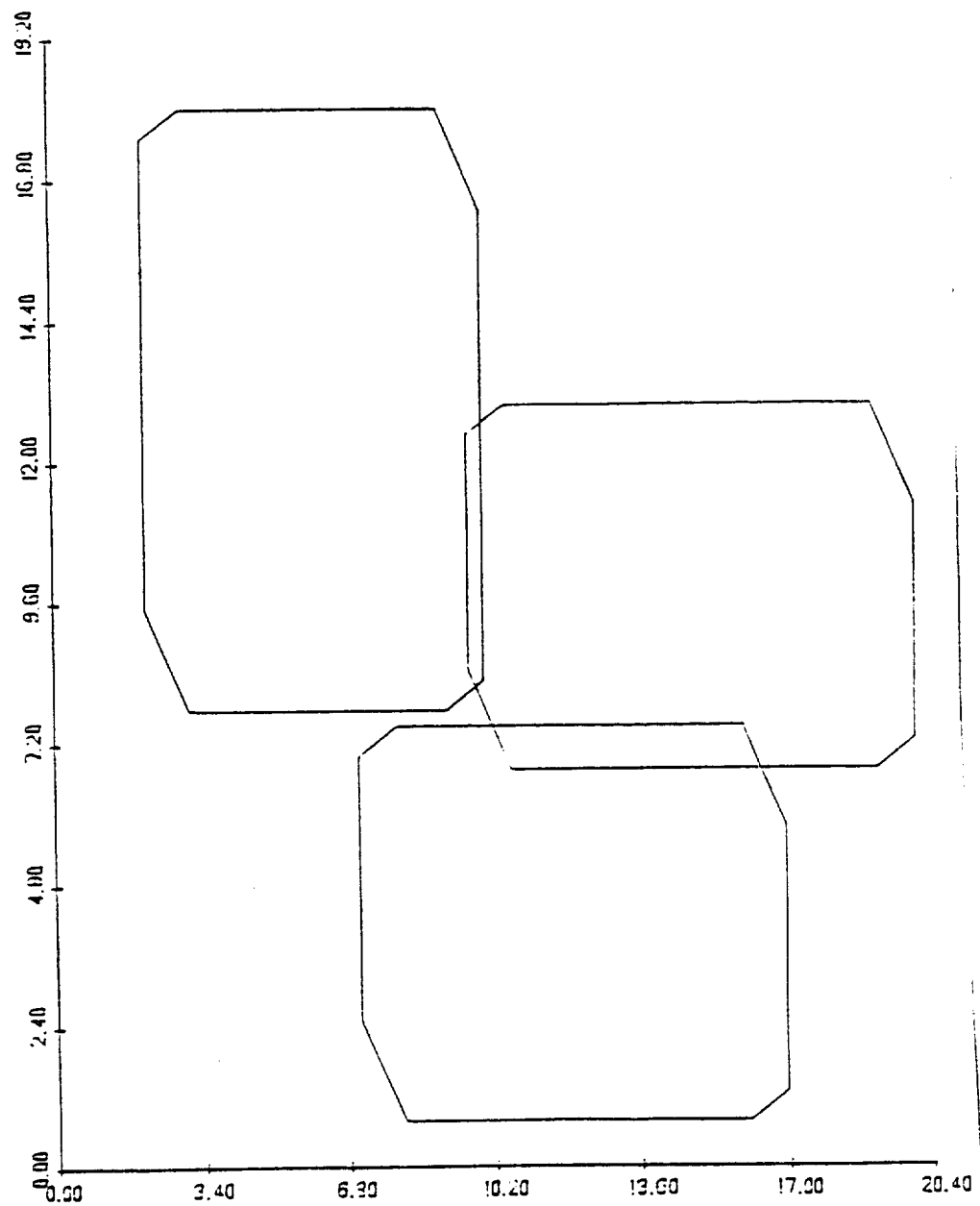


Fig. 2.3.10 A rotational GSpace with $\frac{2\pi}{3}$ sliced

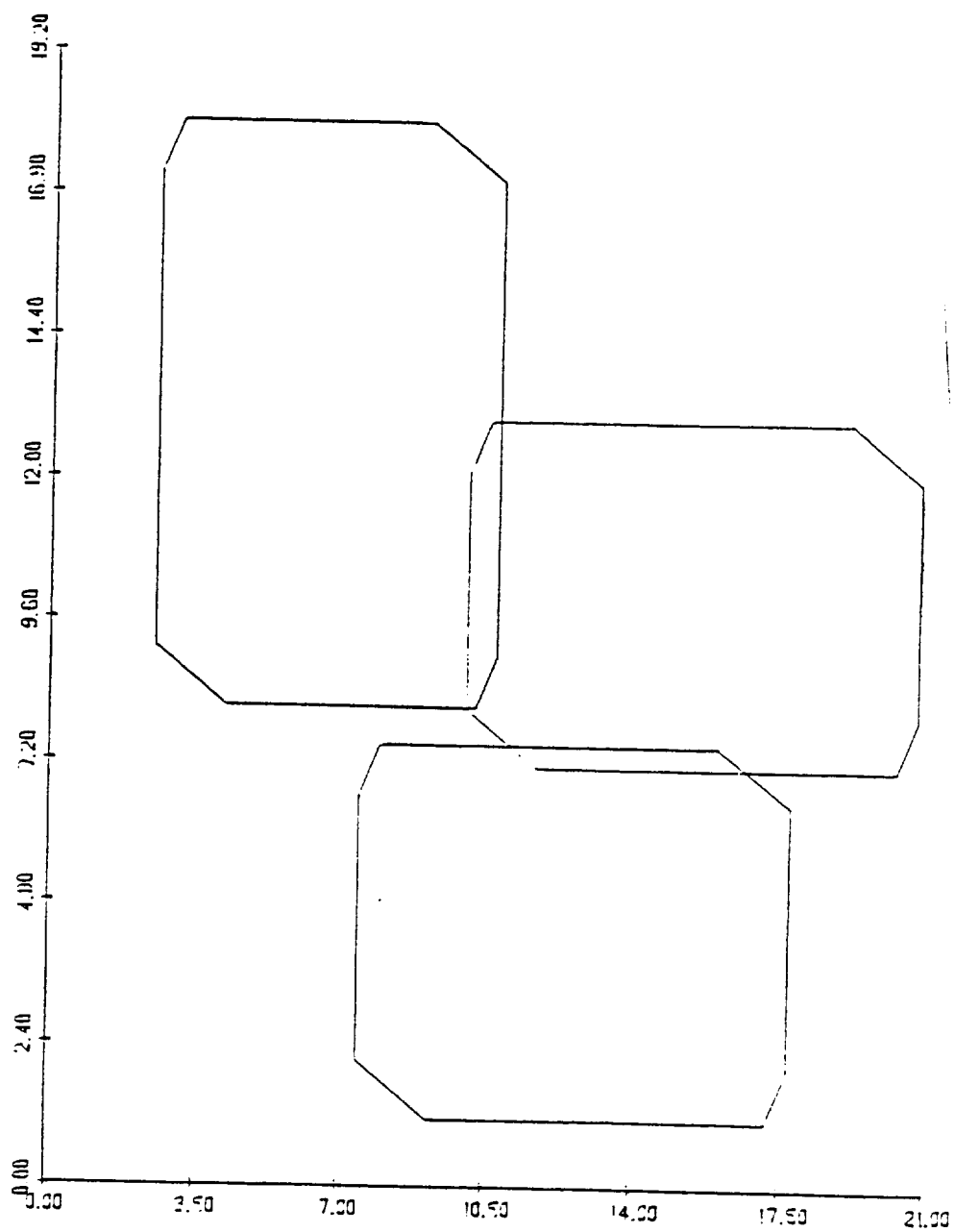


Fig. 2.3.11 A rotational GSpace with $\frac{5\pi}{6}$ sliced

2.4 The VGraph Algorithm

Consider the problem for a moving object to find the collision-free shortest path from the *Start* to the *Goal*. It is desirable that a method represent the *Grown Space Obstacles* with a graph in order to find the shortest path from *Start* and *Goal*. The important property of this path is that it is composed of a straight line joining the *start* point and the *goal* point via a possibly empty sequence of vertices of obstacles [28]. The undirected graph [28] is denoted by $VG(N, L)$ where N is the union of S , G and all the obstacle's vertices. The link set, L , is the set of all the links (N_i, N_j) such that a straight line exists connecting the i^{th} element of N to the j^{th} without overlapping any obstacles. The graph $VG(N, L)$ is thus called the *visibility graph* (*VGraph*) of N , since the connected vertices in the graph can see each other. The *VGraph* for Workspace A is shown in Fig. 2.4.1. To build the *VGraph*, first construct the *Grown Space Obstacles* and then detect the interference of its vertices. If not interfered, the vertice is sent to the set of versible vertices. The *VGraph* can be built by this set of the visible vertices.

The *VGraph Algorithm* requires that the moving object be a point while the obstacles are the forbidden regions for the position of that point [28]. If the moving object is not a point, a new set of obstacles must be computed which are the forbidden regions of some reference point on the moving object. These new obstacles must describe the locus of positions of this reference point which would cause a collision with any set of the original obstacles. The method to build the *Grown Space Obstacles* was described in the previous section. This *VGraph Algorithm* could be applied to find the collision-free shortest path in 2D.

The VGraph Algorithm

1. Build the *Grown Space Obstacles*.
2. Find the visible vertices of the *Grown Space Obstacles*.
3. Build the *VGraph* with the visible vertices.
4. Search the *VGraph* by the graph search algorithm.

It is known that the different initial configuration of a moving object makes the different *GSpace Obstacles*, which result in the different *VGraph*. Since the rotation of a moving object can change its initial configuration, each *VGraph* should be built for its rotation of the moving object. Consider the θ sliced rotation of the moving object. Since the moving object can rotate by θ , the number of its *VGraph*, ξ , can be figured out, where $0 \leq \eta \cdot \theta \leq 2\pi$ and $\eta = 1, 2, \dots, \xi$. For each η ,

build its *VGraph* and construct a set of vertices for all vertices for its *VGraph*. Find the visible vertices from this set of vertices by detecting the interferences. This set of visible vertices can build the *VGraph* with the sliced rotation of a moving object. The algorithm is the following:

Procedure BuildVGraph(var VGraph, List)

Comments

- List is a linked list for the set of visible vertices.
- VGraph is ($n \times n$) array to store the cost between two visible vertices and ∞ means that two vertices are invisible.

Begin

From \leftarrow List

To \leftarrow List

VGraph $\leftarrow \infty$

while From \neq nil do {

if not DeadNode(List, From \uparrow .Node)

then {

To \leftarrow List

while To \neq nil do {

if (From = To) or

DeadNode(List, From \uparrow .Node) or

DeadNode(List, To \uparrow .Node) or

Interference(List, From \uparrow .Node, To \uparrow .Node)

then { do nothing. }

else VGraph \leftarrow cost between two visible
vertices

To \leftarrow To \uparrow .Next } }

From \leftarrow From \uparrow .Next }

End

The shortest path from the *start* to the *goal* in this *VGraph* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles*. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory and the complexity of the *VGraph*,

which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. Therefore, the *Recursive Compensation Algorithm* in section 2.7 is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D.

[Problem Statement 2.4] Consider the problem, shown in Fig. 2.2.1, assuming that the objects are polyhedrons and their visual informations are available and they are represented by vertices. Find the collision-free shortest distance from *Start* to *Goal* with $\frac{\pi}{4}$ sliced rotation.

The programming list of this simulation is available in the Appendix A. The following result for the Problem Statement 2.4 comes from the file [PATH] in Appendix B.

Table 2.4.1 Simulation result of the *VGraph* algorithm.

The shortest path is calculated by the <i>VGraph</i> Algorithm.			
Start Node = 1, Goal Node = 27,			
Path represented by internal nodes: 1 \rightarrow 16 \rightarrow 6 \rightarrow 27			
From	To	Cost	Rotation
Start in 0 sliced	A_3 in $\frac{\pi}{2}$ sliced	4.125	$\frac{\pi}{2}$
A_3 in $\frac{\pi}{2}$ sliced	B_1 in $\frac{\pi}{2}$ sliced	5.025	0
B_1 in $\frac{\pi}{2}$ sliced	Goal in 0 sliced	4.031	$-\frac{\pi}{2}$
The total cost between Start and Goal = 13.179			

Fig. 2.4.2 shows the collision-free shortest path with $\frac{\pi}{4}$ sliced rotation by the *VGraph* Algorithm. The path with $\frac{\pi}{4}$ sliced rotation has 13.179 *Euclidean* distance, while the path without sliced rotation has 25.452 *Euclidean* distance. The path segment with sliced rotation is described in the Table 2.4.1, the path segment without sliced rotation is $\{Start \rightarrow C_1 \rightarrow B_3 \rightarrow B_2 \rightarrow Goal\}$. Hence, the sliced rotation of the moving object can shorten the *Euclidean* distance. However, there is a trade off between accuracy and speed. If the small sliced rotation is considered, then the better approximation to the shortest path can be obtained, but more memory space to store each *VGraph* is needed.

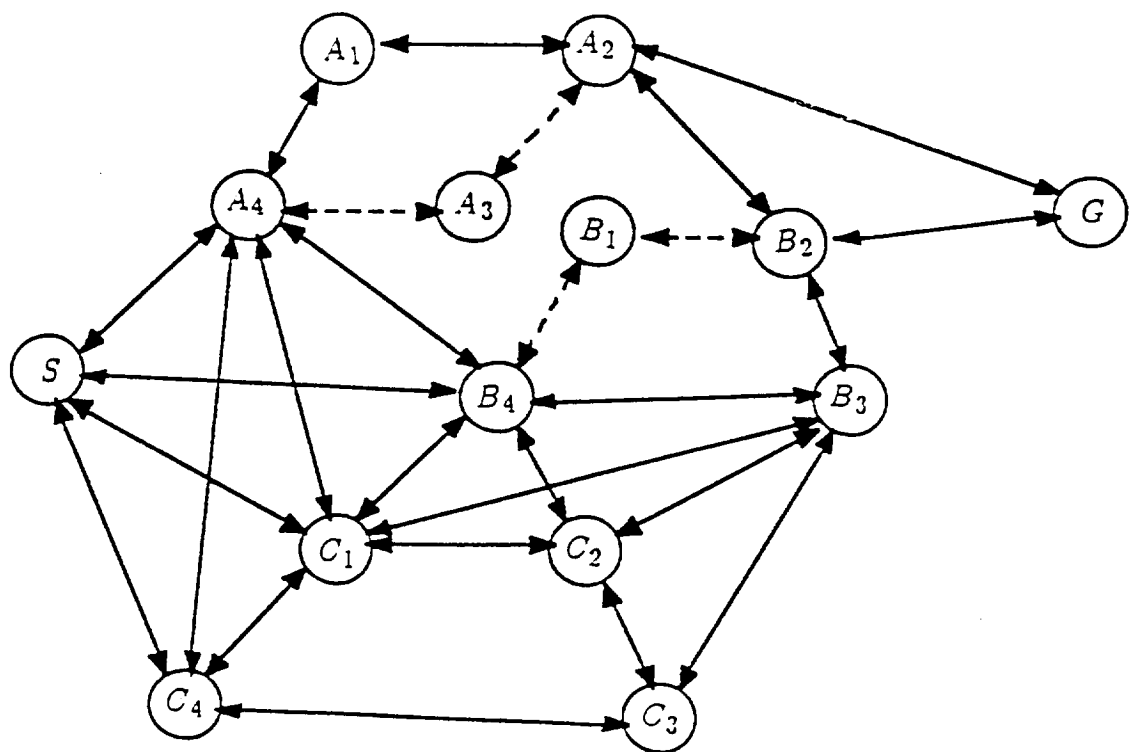


Fig. 2.4.1 A VGraph for Workspace A

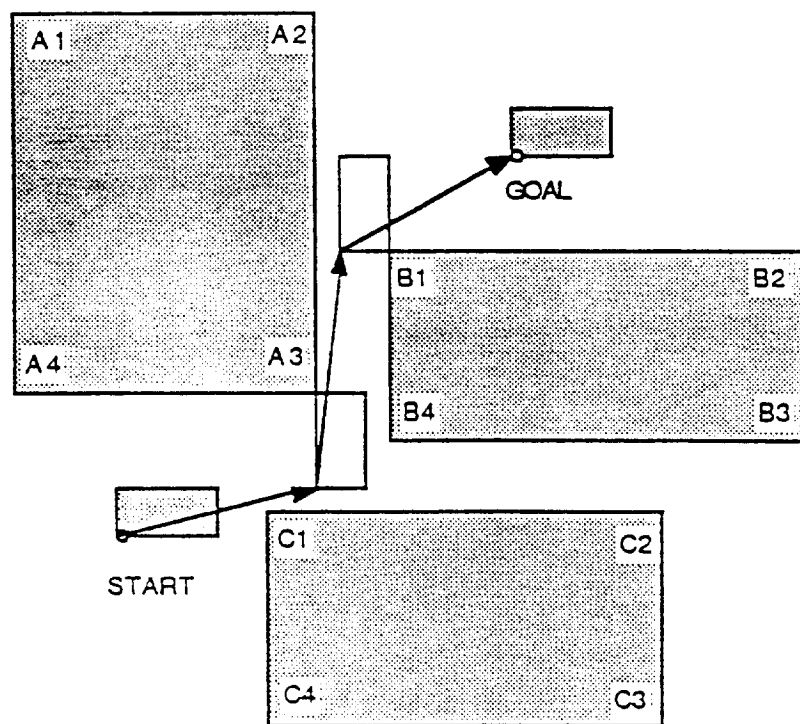


Fig. 2.4.2 The collision-free shortest path for Workspace A

2.5 The Graph Search Algorithm

There are a number of control strategies for finding a path through a graph. The fundamental control problem is to select an appropriate database and the applicable rules to apply in the search for a satisfactory path. The path finding problem has usually been approached in one of two ways [22], the *Mathematical Approach* and the *Heuristic Approach*. The *Mathematical Approach* deals with properties of abstract graphs and with algorithms that follow an orderly examination of the nodes of a graph to find the minimum cost path. The *Heuristic Approach*, on the other hand, typically uses special knowledge about the problem. The efficiency with which a path is found increases as the knowledge becomes closer to being complete. An important point to note is that the *Heuristic Approach* generally is not able to guarantee that the minimum cost path will be found.

General Graph Search Algorithm [61]

1. Put the *START* node on *OPEN*.
2. If *OPEN* is empty, exit with failure.
3. Select a node *n* from *OPEN* and put it on *CLOSE*.
4. Expand *n* and put some of its successors in *OPEN* with a pointer back to *n*. If any of these successors is a *GOAL* node, exit with the solution by tracing back its pointers.
5. Go to step 2.

Pearl [52] describes the main features of the different graph search algorithms as *Hill-climbing*, *Depth-first*, *Backtracking*, *Backmarking*, *Breadth-first*, *Uniform-cost* and *Best-first*. While sharing this common framework, the algorithms differ at least in one of the following points:

- The number of successors generated
- The node from *OPEN* selected for expansion
- The particular management strategies used for cleaning up *CLOSED*

The following remarks [52] should be considered to select a search algorithm for path planning purpose. First, optimality is a concept which seems to be opposed to time and storage efficiency. The only algorithm in which these concepts are somehow compatible is the *A* Algorithm*, as long as a good enough heuristic evaluation function is used. Second, the algorithms in which the goodness of the solution can be established are the *breadth-first* and the *uniform-cost* ones. In most cases this has a high computational cost. Third, to reduce the time and storage re-

quirements of the search algorithms, it is necessary to make them use information which is beyond the pure graph structure. *Hill-climbing*, *backmarking* and *best-first* are the search algorithms which can incorporate information about the particular domain. There are, so, three facts which strongly recommend *best-first* algorithms as the most appropriate ones for path planning purposes: their adequacy for incorporating information belonging to the particular domain being dealt with, their capability of converging to either optimal or non-optimal solutions and their time and storage efficiency. These are the main reasons [61] that led us to select *best-first* algorithms as the most appropriate for path planning. The *A* Algorithm* is probably the most widely used *best-first* graph search procedure. The reason for its success [61] lies in its *simplicity*, its *generality*, the *optimality* of its solutions, and the fact that if its *h* function is *admissible* (that is, it never overestimates the cost of a subpath), the *A* Algorithm* is *optimal* [47]. For example, the criterion used in the *VGraph* can taken into account the distance to be travelled in the *Configuration Space* but also the costs assigned to the change of speed [16].

The total estimate $\hat{f}(n)$ is an estimate of the cost of a minimal cost path from *s* to *g* node constrained to go through node *n*, and can be expressed as $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$, where $\hat{g}(n)$ is an estimate of the cost *g*(*n*) of a minimal cost path from *s* to *n* and $\hat{h}(n)$ is an estimate of the cost *h*(*n*) of a minimal cost path from *n* to a goal node. $\hat{g}(n)$ is constructed step by step by the algorithm, whereas $\hat{h}(n)$ is obtained from the heuristic information.

A* Algorithm [41]

1. Put the start node *s* on a list called *OPEN*. Set $\hat{g}(s) \leftarrow 0$ and $\hat{f}(s) \leftarrow \hat{h}(s)$.
2. If *OPEN* is empty exit with failure; otherwise continue.
3. Remove from *OPEN* that node *n* whose \hat{f} value is smallest and put it on a list called *CLOSED*. (Resolve ties for minimal \hat{f} values arbitrarily, but always in favor of any goal node.)
4. If *n* is a goal node, exit with the solution path obtained by tracing back through the pointers; otherwise continue.
5. Expand node *n*, generating all of its successors. (If there are no successors, go to Step 2.) For each successor n_i , compute $g_i \leftarrow \hat{g}(n) + c(n, n_i)$.
6. If a successor n_i is not already on either *OPEN* or *CLOSED*, set $\hat{g}(n_i) \leftarrow g_i$ and $\hat{f}(n_i) \leftarrow g_i + \hat{h}(n_i)$. Put n_i on *OPEN* and direct a pointer from it back to *n*.
7. If a successor n_i is already on *OPEN* or *CLOSED* and if $\hat{g}(n_i) > g_i$, then update it by setting $\hat{g}(n_i) \leftarrow g_i$ and $\hat{f}(n_i) \leftarrow g_i + \hat{h}(n_i)$. Put

n_i on *OPEN* if it was on *CLOSED* and redirect to n the pointer from n_i .

8. Go to Step 2.

It is possible to prove [47] that if, for every node n , $\hat{h}(n)$ is a lower bound on the cost $h(n)$ of a minimal cost path from node n to a goal node, then the *A* Algorithm*, is *admissible*, i.e. it always finds an optimal path. Moreover, it is possible to simplify the *A* Algorithm* by making a further assumption on the estimate \hat{h} : for any two nodes m and n which are connected by an *arc*(m, n) we have $\hat{h}(m) - \hat{h}(n) \leq c(m, n)$. This assumption is called the *consistency assumption* and its meaning is that, by moving from a node to any successor, we must always have a better estimate. Therefore the *A* Algorithm* with the consistency assumption expands fewer than N nodes and, hence, it runs in $O(N)$ steps [41].

Theorem 2.4.1 [41] For all N there exists a search graph G_N of size N , with positive costs and estimates which are lower bounds ($\hat{h}(n) \leq h(n)$ for each n), on which the *A* Algorithm* runs for $O(2^N)$ steps.

Martelli [41] presents the *B Algorithm* to modify the *A* Algorithm* in order to improve its behaviour with nonconsistent estimate. The *B Algorithm* is thus a simple variant of the *A* Algorithm* and can be obtained from it by substituting steps (1) and (3) with the following steps:

- 1'. Put the start node s on a list called *OPEN*. Set $\hat{g}(s) \leftarrow 0$, $\hat{f}(s) \leftarrow \hat{h}(s)$, $F \leftarrow 0$.
- 3'. If there are some nodes in *OPEN* with $\hat{f} < F$, select among them the node n whose \hat{g} value is smallest; otherwise, select the node n in *OPEN* whose \hat{f} value is smallest and set $F \leftarrow \hat{f}(n)$. (Resolve ties arbitrarily, but always in favor of any goal node.) Remove n from *OPEN* and put it on a list called *CLOSED*.

Theorem 2.4.2 [41] Given any search graph G of size N , with positive costs and estimates which are lower bounds on the minimal cost ($\hat{h}(n) \leq h(n)$ for each n), then the *B Algorithm* runs on it for at most $O(N^2)$ steps.

Theorem 2.4.3 [41] Let G be any search graph with positive costs and estimates which are lower bounds on the minimal cost ($\hat{h}(n) \leq h(n)$ for each node n). Then, if the *A* Algorithm* and the *B Algorithm* resolve ties in the same way, the *B Algorithm* does not expand more nodes than the *A* Algorithm*.

Theorem 2.4.3 assures us that the *B Algorithm* can always be used in place of the *A* Algorithm* without having any loss in efficiency. In particular, when searching trees or graphs with a consistent estimate, both algorithms will have the same behaviour, but, with a nonconsistent estimate, the *B Algorithm* will, in general, have a much better behaviour than the *A* Algorithm*.

2.6 The Orthogonal Projection Method

Most of the work in path planning will be done in the field of building the *Configuration Space Obstacles* rather than searching graph. Therefore, it is clear that the representation of the objects [4] plays a major role in determining the feasibility and performance of any intersection or collision detection method using that representation. Lozano-Pérez [29] [30] shows that algorithms for computing the *Grown Space Obstacles* in 2D have time complexity $O(v)$, and the algorithms for computing the *Grown Space Obstacles* in 3D have time complexity $O(v^2 \log v)$, where v is the total number of vertices. Considering time complexity, it is much better to find a collision-free path projected in 2D rather than in 3D.

Some classes [3] [63] of three-dimensional objects, which are of rigid solids of uniform thickness, can be described by the line drawings of three orthogonal projections onto the two-dimensional planes. For these classes of objects, the three-dimensional model can be reconstructed from the three-dimensional objects. However, for other classes of three-dimensional objects [63], the exact reconstruction from line drawings of three orthogonal projections cannot be performed. However, a maximal volume that encloses the volume of the object could be reconstructed. If a three-dimensional point does not collide with the maximal volume, it would not collide with the true object. This leads to a sufficient condition [63] for collision checking, as stated in the following Lemmas.

Lemma 2.6.1 [63]: If the projection of a three-dimensional point is outside the area of the two-dimensional projection of a three-dimensional object in one or more of the three orthogonal subspaces, the three-dimensional point is guaranteed to be outside the volume of the three-dimensional object in the three-dimensional space.

Lemma 2.6.2 [63]: If the projection of some three-dimensional path for the reference point of a three-dimensional moving object is collision free in one or more of the orthogonal projected spaces, then the three-dimensional moving object is collision-free along the three-dimensional path in the three-dimensional space.

Since the unnecessary obstacles, for the Findpath problem in 3D, can be avoided by Lemma 2.6.1 and Lemma 2.6.2, the *Orthogonal Projection Method* can simplify the *VGraph*. So, the *Orthogonal Projection Method* has some advantages. It can shorten the graph search time as well as it can save the memory space to store the *Grown Space Obstacles* and *VGraph*. And advantage is related with its representation. The *Grown Space Obstacles* in 3D can be represented and built by three *Grown Space Obstacles* in 2D. Three processors are assigned for this job and they work so simultaneously that the parallel processing

can be expected. Therefore, the *Orthogonal Projection Method* can save more time in building the *Grown Space Obstacles* than any other algorithms that work sequentially. The *Orthogonal Projection Method* has the following steps in turn:

1. Project Objects in 3D onto the projection spaces.
2. Build the *Grown Space Obstacles* in 2D.
3. Select the necessary *Grown Space Obstacles* by Lemma 2.6.2.
4. Reconstruct the *Grown Space Obstacles* in 3D.

However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles*. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. Therefore, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D.

[Problem Statement 2.6] Build the *Grown Space Obstacles* in 3D by using the *Orthogonal Projection Method*, assuming that the object in 3D is a polyhedron, shown in Fig. 2.6.1. The object has the following vertices; $P_1(x_1, y_1, z_1)$, $P_2(x_1, y_2, z_1)$, $P_3(x_2, y_2, z_1)$, $P_4(x_2, y_1, z_1)$, $P_5(x_1, y_1, z_2)$, $P_6(x_1, y_2, z_2)$, $P_7(x_2, y_2, z_2)$, $P_8(x_2, y_1, z_2)$, where $x_1 = 7, y_1 = 5, z_1 = 3, x_2 = 14, y_2 = 10, z_2 = 12$.

Fig. 2.6.2 describes three Orthogonal Projections of Workspace D. Fig. 2.6.3 describes the *Grown Space Obstacles* in 2D. Fig. 2.6.4 describes the reconstruction of the *Grown Space Obstacles* in 3D. Fig. 2.6.5 describes the *Grown Space Obstacles* of Workspace D.

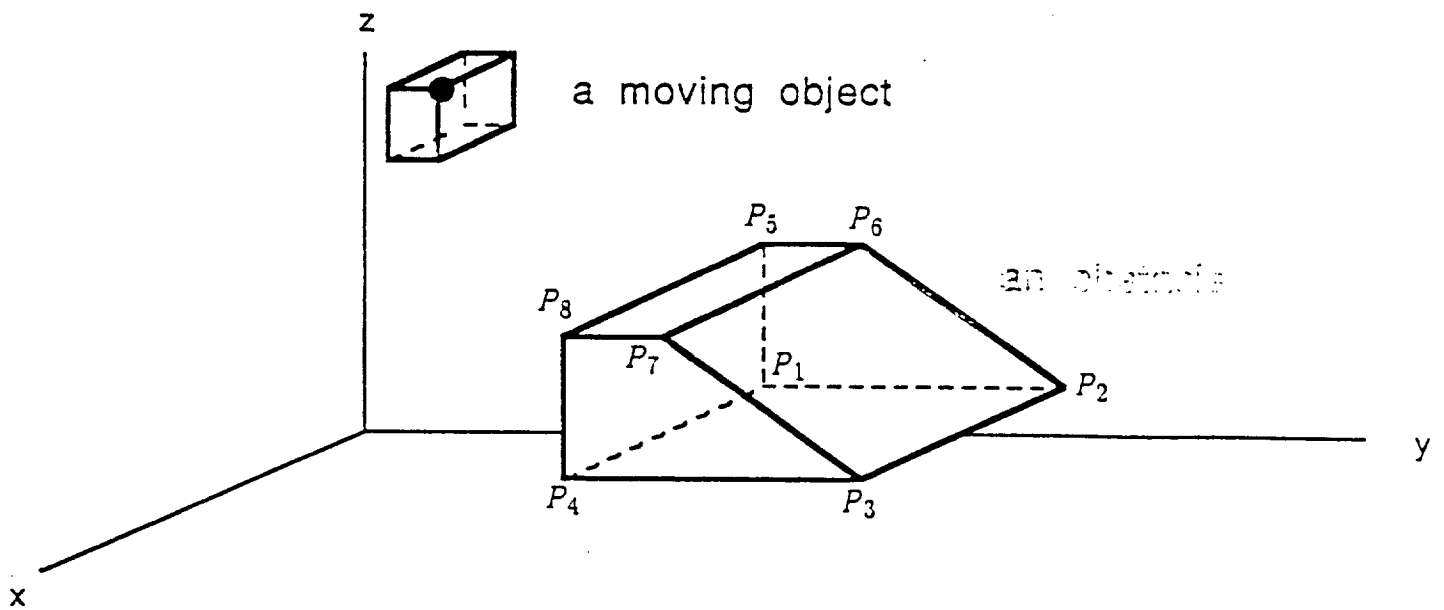


Fig. 2.6.1 A description of Workspace D

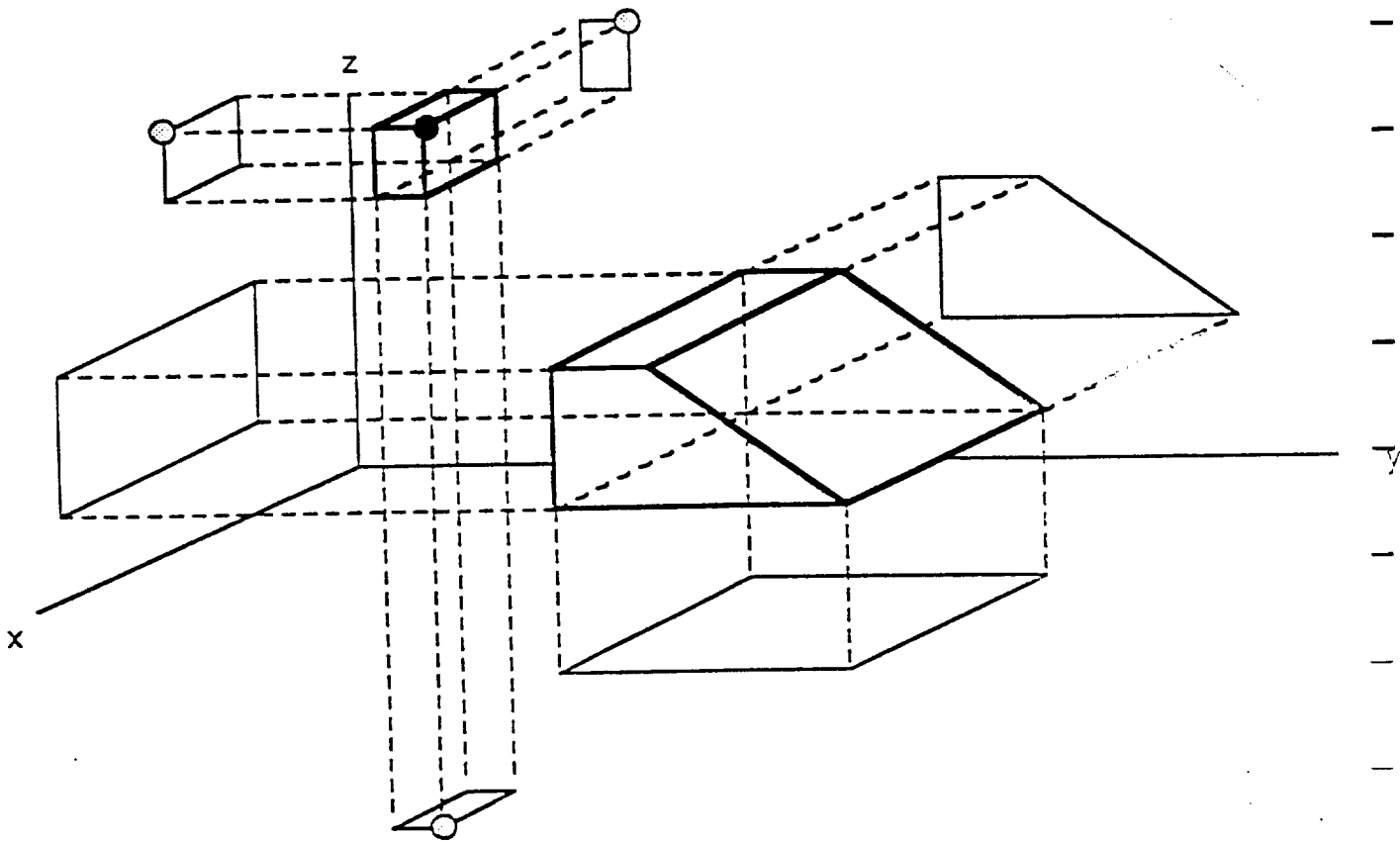


Fig. 2.6.2 A description of three Orthogonal Projections

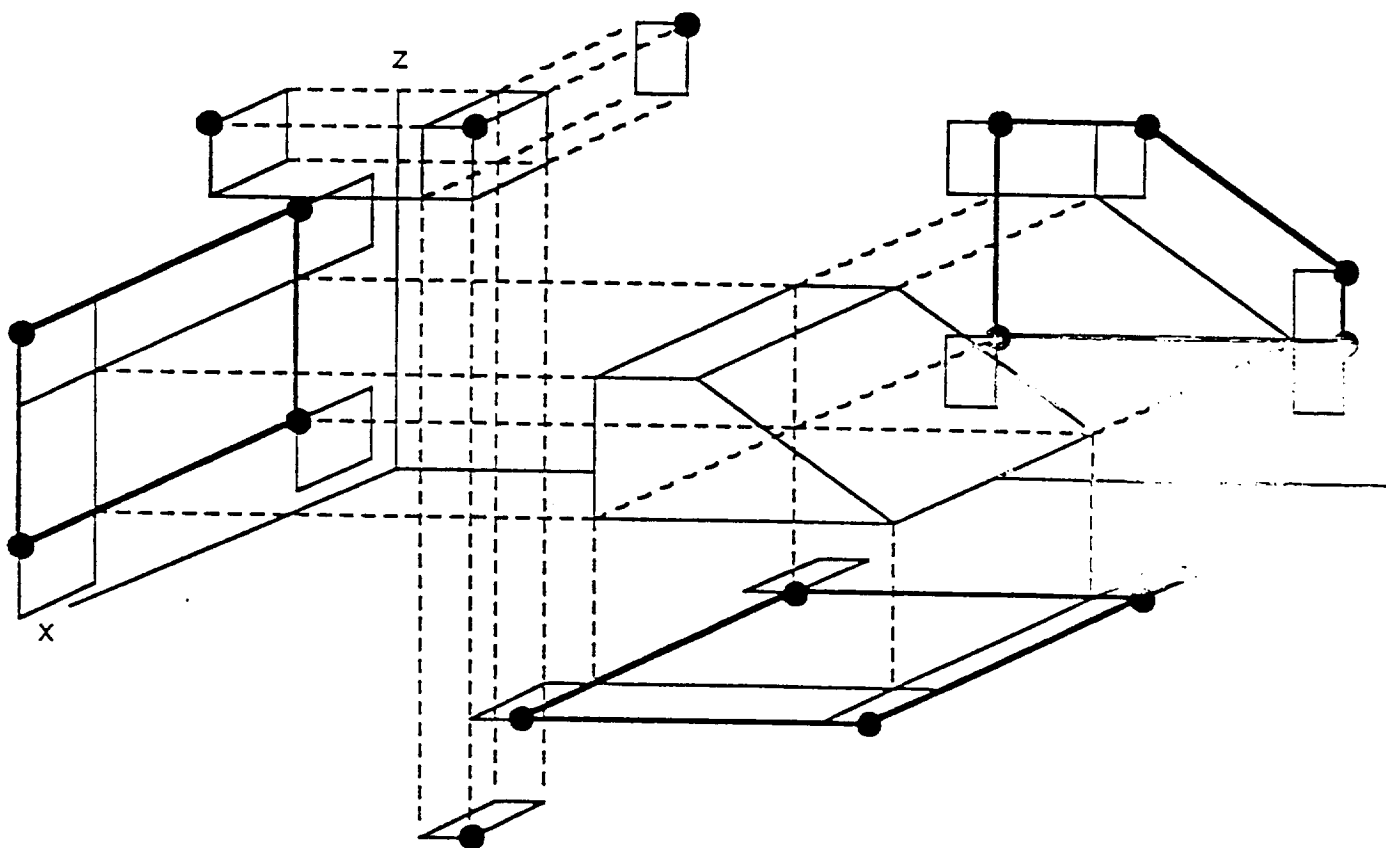


Fig. 2.6.3 A description of Grown Space Obstacles in 2D

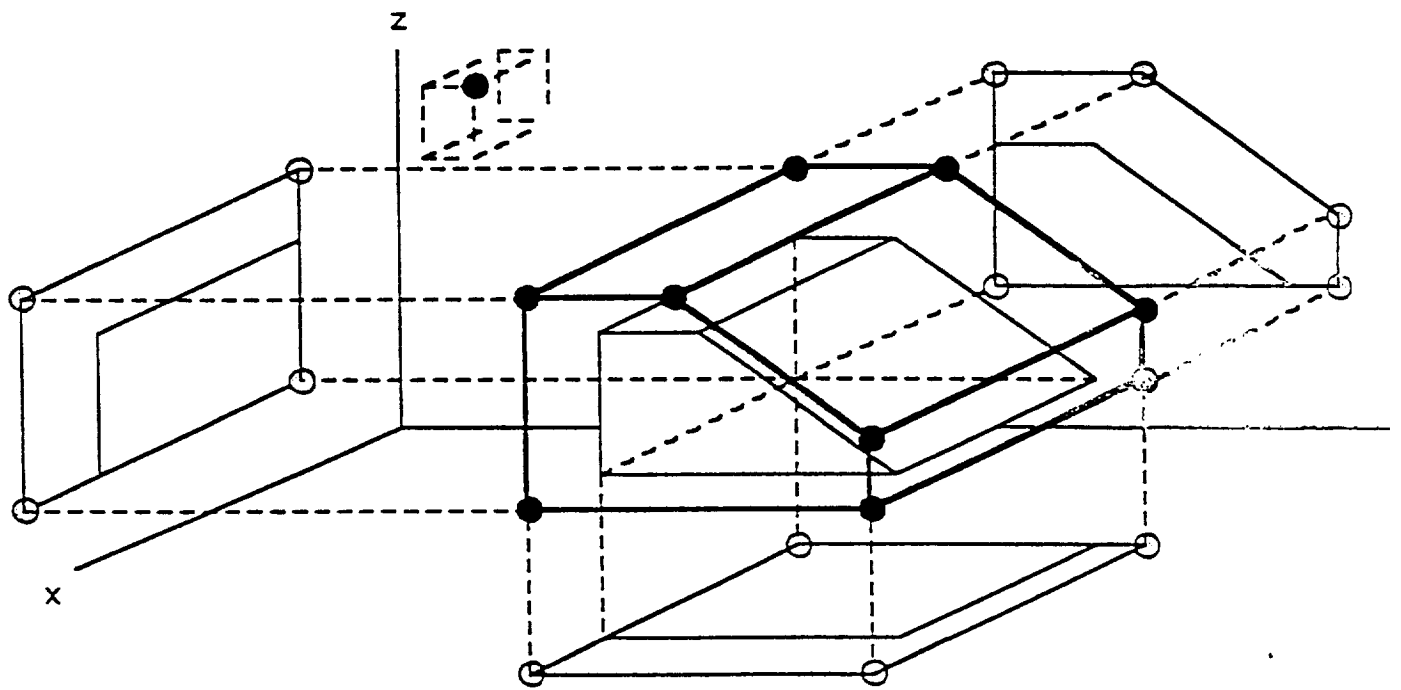


Fig. 2.6.4 A reconstruction of Grown Space Obstacles in 3D

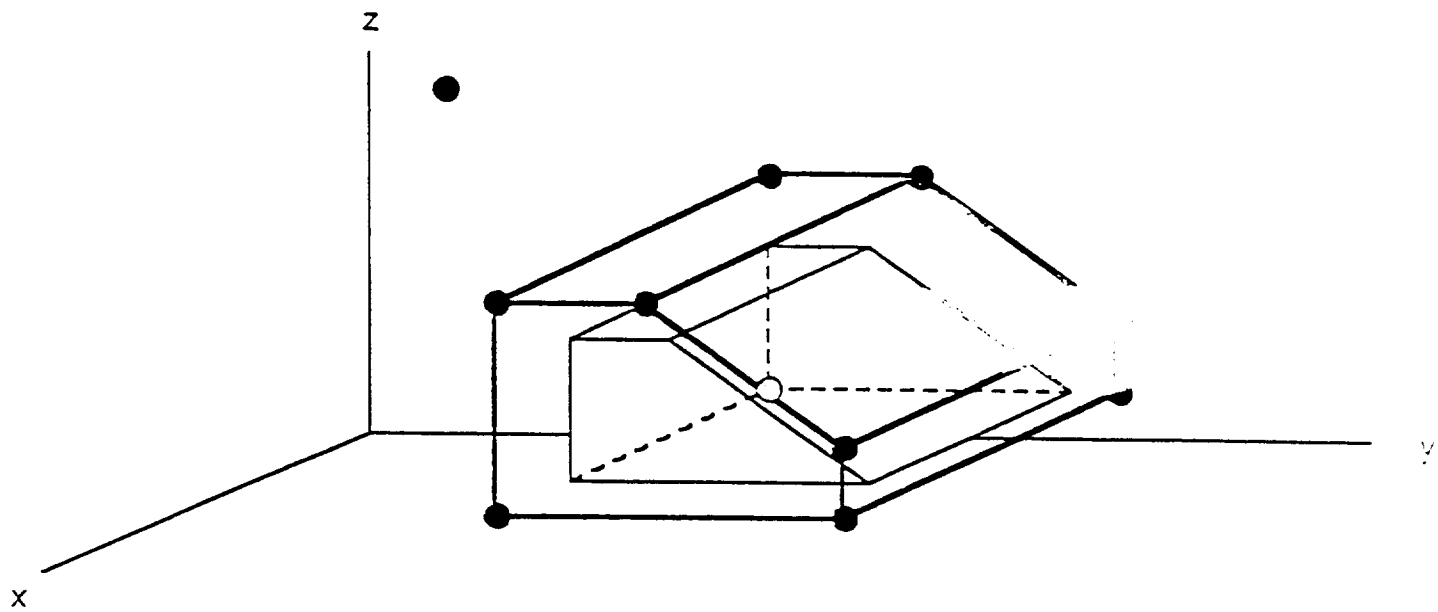


Fig. 2.6.5 The Grown Space Obstacles of Workspace D

2.7 The Recursive Compensation Algorithm

The shortest path from the start to the goal in this *VGraph Algorithm* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles*. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long computational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. Therefore, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D.

Fig. 2.7.1 describes the path calculated by the *VGraph Algorithm* in 3D. The shortest path in 3D may involve going through points on the edges of the *Grown Space Obstacles*. Therefore, the path by *VGraph Algorithm* is not guaranteed to be the shortest path, since the path by the *VGraph Algorithm* involves going through points only on the vertices of the *Grown Space Obstacles*. Fig. 2.7.2 describes the first compensation for the intermediate nodes. Since we assume that all the obstacles are polyhedrals, we are interested in the $z_i(k)$, $i = 1, 2$ and $k = 0, 1, 2, \dots$. The subscript i indicates the intermediate node and k indicates the number of recursive compensation. This problem is related with minimizing the *Euclidean* distance between two nodes in 3D within some constraints. Fig. 2.7.2 describes the second compensation for the intermediate nodes and Fig. 2.7.3 describes the third compensation for them. Fig. 2.7.4 describes the final path in the *Recursive Compensation Algorithm*, i.e., the better approximation to the shortest path in 3D.

RCA (Recursive Compensation Algorithm)

Procedure RCA(NodeSet, ϵ)

Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \dots, N_{n-1}\} \cup \{G\}$ searched by *VGraph* where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, \dots , $G(x_n, y_n, z_n)$
The NodeSet is represented by the linked list.
- ϵ = a permissible error
- Function *Distance* will calculate the *Euclidean* distance through NodeSet.
- Procedure *Compensate* will find the compensated nodes and will return the set of these nodes.

Begin

```

     $D_1 = \text{Distance}(\text{NodeSet})$ 
    Compensate(NodeSet)
     $D_2 = \text{Distance}(\text{NodeSet})$ 
    If  $|D_1 - D_2| < \epsilon$ 
        Then Return(NodeSet)
        Else RCA(NodeSet,  $\epsilon$  )

```

End

Function Distance(NodeSet)

Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \dots, N_{n-1}\} \cup \{G\}$ searched by *VGraph* where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, \dots , $G(x_n, y_n, z_n)$
The NodeSet is represented by the linked list.
- $\#$ of NodeSet for *Distance* ≥ 2
- Function *ED* will find the *Euclidean* distance between the first node and the second node in NodeSet.

Begin

```

    If  $\#$  of NodeSet = 2
        Then Distance = ED(NodeSet)
        Else Distance = ED(NodeSet)
            + Distance(NodeSet - {FirstNode})

```

End

Procedure Compensate(NodeSet)

Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \dots, N_{n-1}\} \cup \{G\}$ searched by *VGraph* where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, \dots , $G(x_n, y_n, z_n)$
The NodeSet is represented by the linked list.
- # of NodeSet for Compensate ≥ 3
- Procedure *Reset* will take the first 3 nodes in NodeSet and replace the second node of the 3 nodes in NodeSet in order to get the set of compensated nodes.

Begin

 If # of NodeSet = 3

 Then Reset(NodeSet)

 Else Reset(NodeSet) \cup Compensate(NodeSet - {FirstNode})

End

Procedure Reset(NodeSet)

Comments

- NodeSet = $\{S\} \cup \{N_1, N_2, \dots, N_{n-1}\} \cup \{G\}$ searched by *VGraph* where $S(x_0, y_0, z_0)$, $N_1(x_1, y_1, z_1)$, $N_2(x_2, y_2, z_2)$, \dots , $G(x_n, y_n, z_n)$
The NodeSet is represented by the linked list.
- # of NodeSet for *Reset* ≥ 3
- $D(d)$ is the *Euclidean* distance function via 3 nodes in 3D.
- Refer to Appendix C [11] to calculate d to satisfy $\frac{\partial D(d)}{\partial d} = 0$.

Begin

 Take out the first 3 nodes in NodeSet.

 Calculate d to satisfy $\frac{\partial D(d)}{\partial d} = 0$.

 If d is on the visible edge,

 Then replace the second in NodeSet with the compensated.

End.

[Problem Statement 2.7] Suppose the following vertices are calculated by the *VGraph Algorithm*; $S(3, 2, 4)$, $N_1(7, 4, 10)$, $N_2(8, 8, 9)$, $G(4, 11, 2)$ shown in Fig. 2.7.1. Calculate the shortest path from the S node to the G node, assuming that the obstacles are polyhedrals.

Define the *convergence ratio* (λ_i)

$$\lambda_i = \frac{y_i(k) - y_i(k-1)}{y_i(k-1) - y_i(k)}$$

where i for intermediate node, k for recursion. Then the *RCA* have the fast convergence ratio from the Theorem 2.7.3. When ϵ is set to 10^{-5} , the *Branch and Bound Method* needs 2301666 milliseconds, however, the *RCA* needs only 416 milliseconds on VAX-11/750. The *Euclidean* distance obtained by the *RCA* is 48% less than that obtained by the *VGraph Algorithm*. The *RCA* is 55,000 times faster than the *Branch and Bound Method* within the same ϵ from the Table 2.7.1. Fig. 2.7.5 describes how fast the *RCA* works. It is proved that the sequences generated by the *RCA* are *Cauchy sequences* by the Theorem 2.7.4. Therefore, the number of recursive compensation could be calculated if ϵ is known. Or ϵ could be calculated if the number of recursive compensation is given. Let's compare the Lozano-Pérez's alleviation method and the *Recursive Compensation Algorithm* to get the same accuracy, ϵ , for the Problem Statement 2.7. Since ϵ is set to 10^{-5} , Lozano-Pérez's alleviation method needs a lot of memory space to store $(2 + 2 \times 8 \times 10^5)$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2 + 2 \times 8)$ vertices for the *VGraph*. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time.

Table. 2.7.1 Euclidean distance and Computing time

Algorithm	Distance	Computing time
VGraph	20.3283	0
Branch and Bound	13.7416	2.301.666
RCA	13.7416	416

The simulation of the *RCA* has been done in 3D. See the *Problem Statement 2.7* and the simulation result in 3D. The convergence *RCA* is proved in 2D for simplicity. We are interested in the convergence of the compensated nodes by the *RCA*, i.e., $\{y_1(k)\}$, $\{y_2(k)\}$.

$$\begin{aligned}
 y_1(k) &= y_0 + \frac{y_2(k-1) - y_0}{x_2 - x_0}(x_1 - x_0) \\
 &= y_0 + \Delta_0 \{y_2(k-1) - y_0\} \\
 &= \Delta_0 y_2(k-1) + (1 - \Delta_0) y_0
 \end{aligned}$$

$$\Delta_0 \equiv \frac{x_1 - x_0}{x_2 - x_0}, 0 < \Delta_0 < 1$$

$$\begin{aligned}
 y_2(k) &= y_1(k) + \frac{y_3 - y_1(k)}{x_3 - x_1}(x_2 - x_1) \\
 &= y_1(k) + \Delta_1 \{y_3 - y_1(k)\} \\
 &= (1 - \Delta_1) y_1(k) + \Delta_1 y_3.
 \end{aligned}$$

$$\Delta_1 \equiv \frac{x_2 - x_1}{x_3 - x_1}, 0 < \Delta_1 < 1$$

Definition 2.7.1 [6] [54] Let $\{x_n\}$ be a sequence of extended real numbers; we define the *limit superior of $\{x_n\}$* to be the extended real number

$$\limsup x_n = \inf_{n \geq 1} \sup_{j \geq n} x_j$$

and the *limit inferior of $\{x_n\}$* to be the extended real number

$$\liminf x_n = \sup_{n \geq 1} \inf_{j \geq n} x_j.$$

Proposition 2.7.1 [6] [54] Let $\{x_n\}$ be a sequence of extended real numbers and set

$$s_n^- = \inf_{j \geq n} x_j$$

and

$$s_n^+ = \sup_{j \geq n} x_j.$$

Then for each n the following hold:

- i. $s_n^- \leq s_n^+$;
- ii. $s_{n+1}^+ \leq s_n^+$;
- iii. $s_{n+1}^- \geq s_n^-$;
- iv. $\liminf x_n \leq \limsup x_n$.

Proof. Conclusions (i)-(iii) are obvious from the definitions, and so we prove only (iv). Define

$$s^- = \sup s_n^- (= \liminf x_n)$$

and

$$s^+ = \inf s_n^+ (= \limsup x_n).$$

Fix an integer n ; observe that if $1 \leq k \leq n$, then by (iii)

$$s_k^- \leq s_n^- \leq s_n^+.$$

On the other hand, if $k > n$, then we may apply (ii) to obtain

$$s_k^- \leq s_k^+ \leq s_n^+$$

and thus s_n^+ is an upper bound for $\{s_k^-\}$. This implies that $s_n^+ \geq s^-$; but n was arbitrary, and thus s^- is a lower bound for $\{s_n^+\}$, showing that $s^- \leq s^+$, as desired.

Theorem 2.7.1 [6] [54] Let $\{x_n\}$ be a sequence of extended real numbers and suppose that $\lim x_n = x_\infty$. If $|x_\infty| < \infty$, then for each $\epsilon > 0$ there is an integer N such that $|x_j - x_\infty| < \epsilon$ whenever $j \geq N$.

Proof. Define $\{s_n^-\}$ and $\{s_n^+\}$ as in Proposition 2.7.1 and fix $\epsilon > 0$. Since $\sup\{s_n^-\} = x_\infty = \inf\{s_n^+\}$, we may choose N_1 and N_2 so large that

$$x_\infty - \epsilon \leq s_{N_1}^-$$

and

$$x_\infty + \epsilon \leq s_{N_2}^+.$$

Setting

$$N = \max\{N_1, N_2\},$$

it follows from Proposition 2.7.1 (ii) and (iii) that for $j \geq N$,

$$x_\infty - \epsilon \leq s_N^- \leq x_j \leq s_N^+ \leq x_\infty + \epsilon,$$

from which $|x_j - x_\infty| \leq \epsilon$ if $j \geq N$.

Definition 2.7.2 [6] [54] Let $\{x_n\}$ be a sequence of real numbers. Suppose, for each $\epsilon > 0$, there is an index N such that $|x_j - x_k| < \epsilon$ whenever $j, k \geq N$. Then the sequence $\{x_n\}$ is said to be a *Cauchy sequence*.

Theorem 2.7.2 [6] [54] For a sequence $\{x_n\}$ of real numbers, the following are equivalent;

- i. $\{x_n\}$ is a *Cauchy sequence*;
- ii. $\{x_n\}$ converges to some real number x_∞ .

Proof. We first show that (i) \implies (ii). By Proposition 2.7.1 (iv), it suffices to argue that $\limsup x_n \leq \liminf x_n$. Fix $\epsilon > 0$ and choose N so large that if $j, k \geq N$, then $|x_j - x_k| \leq \epsilon$. Then in particular, $j, k \geq N$, it follows that $x_j \leq x_k + \epsilon$ and hence

$$\inf_{n \geq 1} \sup_{j \geq n} x_j \leq \sup_{j \geq N} x_j \leq x_k + \epsilon.$$

Now recall that $k \geq N$ was arbitrary, and hence

$$\limsup x_n \leq \sup_{n \geq N} \inf_{k \geq n} x_k + \epsilon = \liminf x_n + \epsilon.$$

Since ϵ was arbitrary, this shows that (i) \implies (ii). For the converse, fix $\epsilon > 0$ and applying Theorem 2.7.1, select an integer N so that $|x_j - x_\infty| \leq \epsilon/2$ if $j \geq N$. Then $j, k \geq N$,

$$|x_j - x_k| \leq |x_j - x_\infty| + |x_\infty - x_k| \leq \epsilon.$$

Theorem 2.7.3 The sequences generated by ~~RCI~~ are ~~mono decreasing~~.

Proof. The n^{th} compensated node $y_1(k)$ can be described by

$$y_1(k) = \Delta_0(1 - \Delta_1)y_1(k-1) + (1 - \Delta_0)y_0 + \Delta_0\Delta_1y_3$$

Take $(n+1)^{\text{th}}$ compensated node $y_1(k+1)$,

$$y_1(k+1) = \Delta_0(1 - \Delta_1)y_1(k) + (1 - \Delta_0)y_0 + \Delta_0\Delta_1y_3$$

Subtract one from the other, then

$$y_1(k) - y_1(k+1) = \Delta_0(1 - \Delta_1)\{y_1(k-1) - y_1(k)\}$$

$$\frac{y_1(k) - y_1(k+1)}{y_1(k-1) - y_1(k)} = \Delta_0(1 - \Delta_1)$$

Since $|\Delta_0(1 - \Delta_1)| < 1$,

$$\left| \frac{y_1(k) - y_1(k+1)}{y_1(k-1) - y_1(k)} \right| < 1$$

Therefore, $\{y_1(k)\}$ is mono decreasing.

The n^{th} compensated node $y_2(k)$ can be described by

$$y_2(k) = \Delta_0(1 - \Delta_1)y_2(k-1) + (1 - \Delta_0)(1 - \Delta_1)y_0 + \Delta_1y_3$$

Take $(n+1)^{\text{th}}$ compensated node $y_2(k+1)$,

$$y_2(k+1) = \Delta_0(1 - \Delta_1)y_2(k) + (1 - \Delta_0)(1 - \Delta_1)y_0 + \Delta_1y_3$$

Subtract one from the other, then

$$y_2(k) - y_2(k+1) = \Delta_0(1 - \Delta_1)\{y_2(k-1) - y_2(k)\}$$

$$\frac{y_2(k) - y_2(k-1)}{y_2(k-1) - y_2(k)} = \Delta_0(1 - \Delta_1)$$

Since $|\Delta_0(1 - \Delta_1)| < 1$,

$$\left| \frac{y_2(k) - y_2(k+1)}{y_2(k-1) - y_2(k)} \right| < 1$$

Therefore, $\{y_2(k)\}$ is mono decreasing.

QED.

Theorem 2.7.4 The sequences generated by RCA are *Cauchy sequences*.

Proof. This theorem can be proved by the *mathematical induction*. By the Theorem 2.7.3, we can get

$$\frac{y_1(2)}{y_1(1)} < 1$$

Substitute $y_1(2) = \Delta_0(1 - \Delta_1)y_1(1) + \Delta_0\Delta_1y_3 + (1 - \Delta_0)y_0$

$$\frac{\Delta_0(1 - \Delta_1)y_1(1) + \Delta_0\Delta_1y_3 + (1 - \Delta_0)y_0}{y_1(1)} < 1$$

$$\{1 - \Delta_0(1 - \Delta_1)\}y_1(1) > (1 - \Delta_0)y_0 + \Delta_0\Delta_1y_3$$

(i) Since $1 - \Delta_0(1 - \Delta_1) > 0$, we can get the following inequality

$$y_1(1) > \frac{1 - \Delta_0}{1 - \Delta_0(1 - \Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1 - \Delta_0(1 - \Delta_1)}y_3$$

(ii) Assume that

$$y_1(k) > \frac{1 - \Delta_0}{1 - \Delta_0(1 - \Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1 - \Delta_0(1 - \Delta_1)}y_3$$

(iii) We should prove that

$$y_1(k+1) > \frac{1 - \Delta_0}{1 - \Delta_0(1 - \Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1 - \Delta_0(1 - \Delta_1)}y_3$$

If the above inequality is hold for any k , the sequence generated by RCA is a *Cauchy sequence* because the sequence by RCA is mono decreasing by the Theorem 2.7.3.

$$y_1(k+1) = \Delta_0(1 - \Delta_1)y_1(k) + (1 - \Delta_0)y_0 + \Delta_0\Delta_1y_3$$

Substitute $y_1(k)$ with the inequality from (ii), then

$$y_1(k+1) > \Delta_0(1-\Delta_1)\left\{\frac{1-\Delta_0}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3\right\} \\ + (1-\Delta_0)y_0 + \Delta_0\Delta_1y_3$$

Therefore,

$$y_1(k+1) > \frac{1-\Delta_0}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_0\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

By the Theorem 2.7.3, we can get

$$\frac{y_2(2)}{y_2(1)} < 1$$

Substitute $y_2(k) = \Delta_0(1-\Delta_1)y_2(k-1) + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1y_3$

$$\frac{\Delta_0(1-\Delta_1)y_2(k-1) + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1y_3}{y_1(1)} < 1$$

$$\{1-\Delta_0(1-\Delta_1)\}y_2(1) > (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1y_3$$

(iv) Since $1-\Delta_0(1-\Delta_1) > 0$, we can get the following inequality.

$$y_2(1) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

(v) Assume that

$$y_2(k) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

(vi) We should prove that

$$y_2(k+1) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

If the above inequality is hold for any k , the sequence generated by RCA is a *Cauchy sequence* because the sequence by RCA is mono decreasing by the Theorem 2.7.3.

$$y_2(k) = \Delta_0(1-\Delta_1)y_2(k-1) + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1y_3$$

Substitute $y_2(k)$ with the inequality from (v), then

$$y_2(k+1) > \Delta_0(1-\Delta_1)\left\{\frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3\right\} \\ + (1-\Delta_0)(1-\Delta_1)y_0 + \Delta_1y_3$$

Therefore,

$$y_2(k+1) > \frac{(1-\Delta_0)(1-\Delta_1)}{1-\Delta_0(1-\Delta_1)}y_0 + \frac{\Delta_1}{1-\Delta_0(1-\Delta_1)}y_3$$

Therefore, the sequences generated by RCA are *Cauchy sequences* by the Theorem 2.7.2 and Theorem 2.7.3, because $\{y_1\}$ and $\{y_2\}$ converge to some real numbers and they are mono decreasing.

QED.

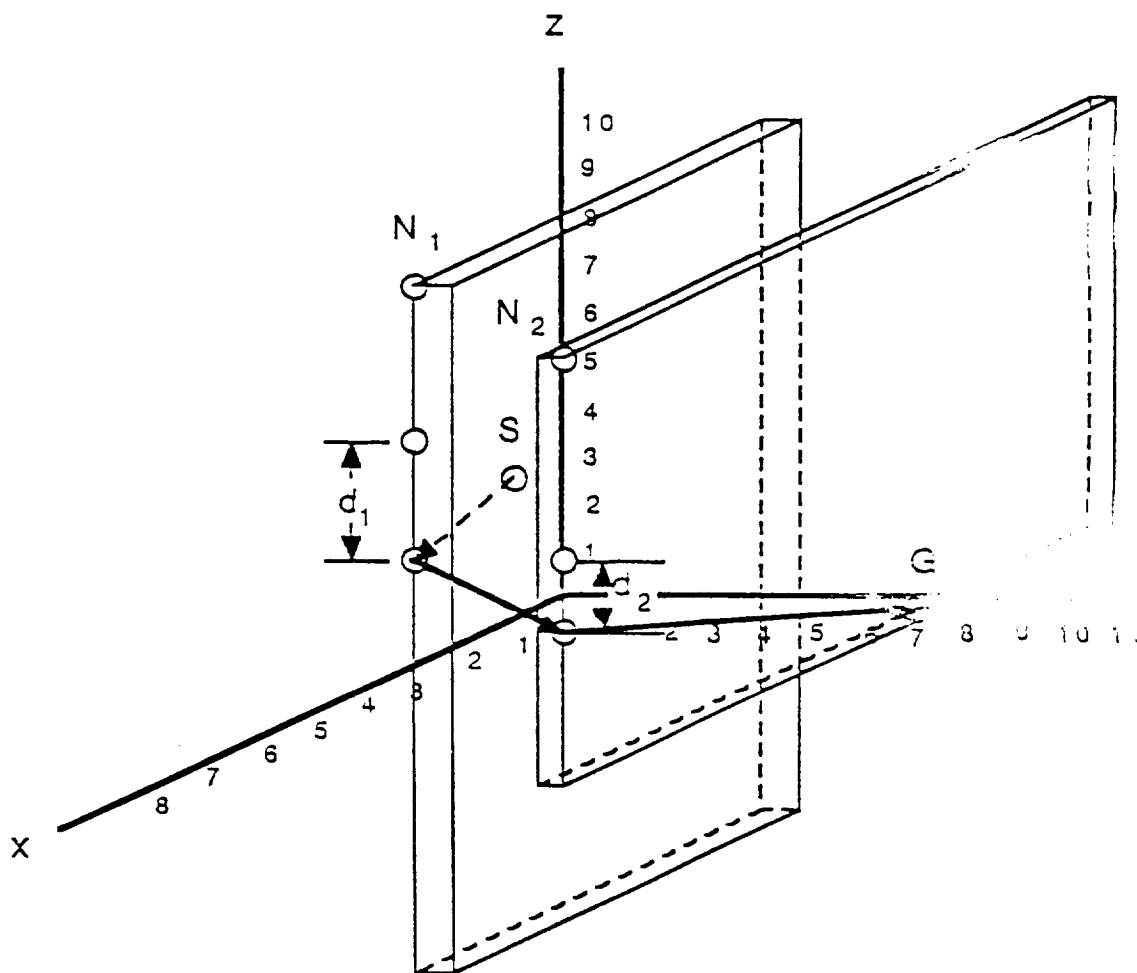


Fig. 2.7.3 The second compensation by the RCA

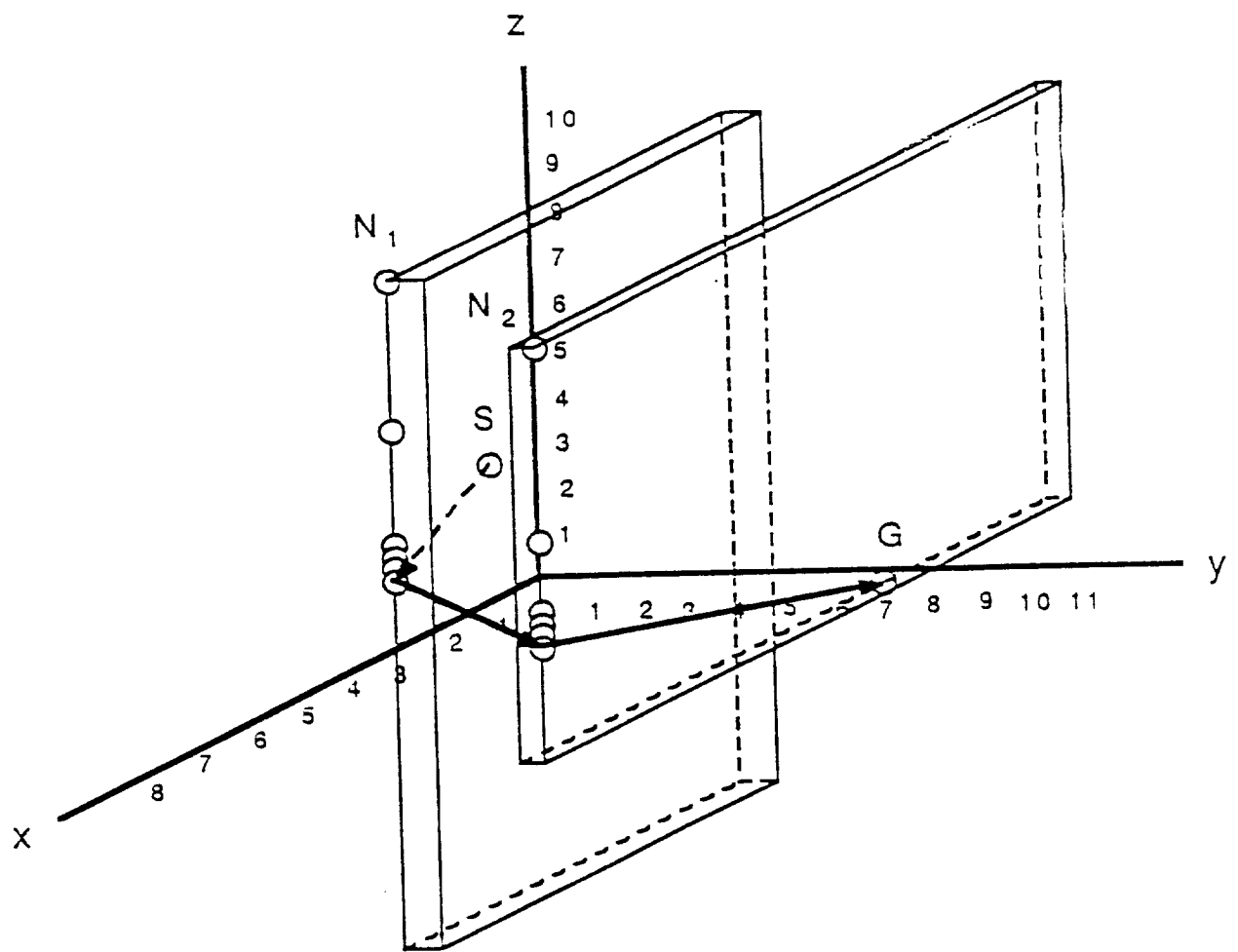


Fig. 2.7.4 The final path by the RCA

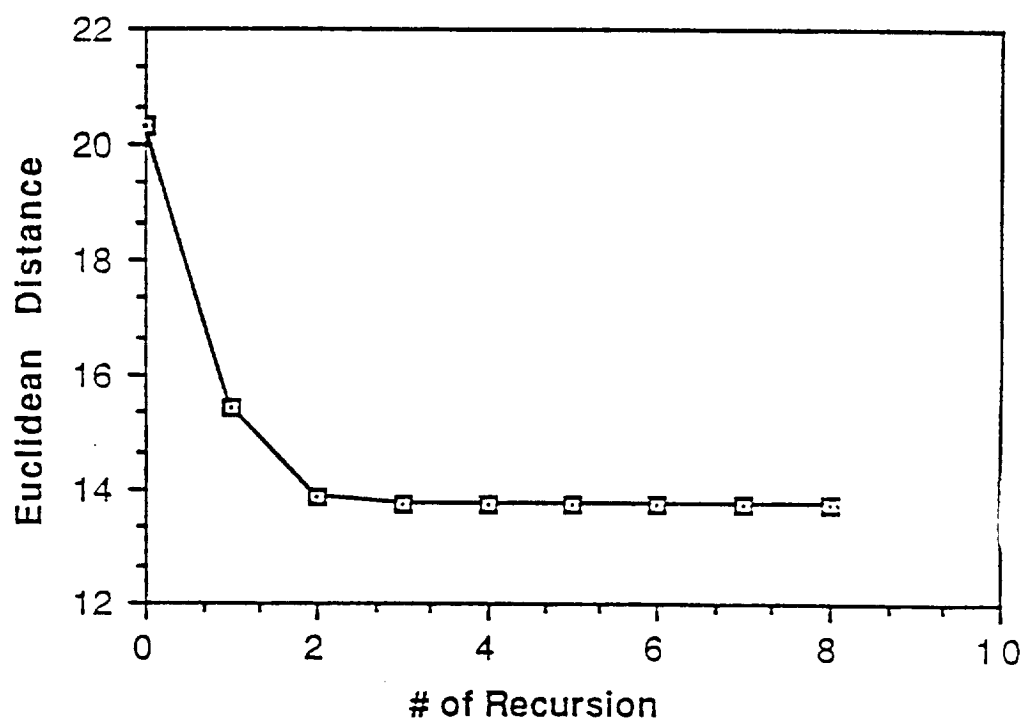


Fig. 2.7.5 The Euclidean distance by the RCA

3. Problem Statement, Preliminary Results and Proposed Work

3.1 Problem Statement

Consider the *VGraph Algorithm* for a moving object to find the collision-free shortest path in a workspace with some obstacles. A lot of work has been done in this field, which has the following design steps:

- Build the *Grown Space Obstacles*.
- Find the visible vertices by detecting interferences.
- Build the *VGraph* with a set of the visible vertices.
- Search the *VGraph* by the graph search algorithm.

The shortest path from the start to the goal in this *VGraph Algorithm* is the shortest path among the obstacles in 2D. However, the path in 3D by the *VGraph Algorithm* [28] [29] whose node set contains only vertices of the *Grown Space Obstacles* is not guaranteed to be the shortest collision free path, because the shortest path may involve going through points on the edges of the *Grown Space Obstacles* in 3D. Lozano-Pérez [29] points out the drawbacks of the *VGraph Algorithm*. The first drawback is related with the rotation of a moving object. Since the *VGraph Algorithm* require moving a object along obstacle boundaries, shortest paths are very susceptible to inaccuracies in the object models. This drawback can be solved by using the *sliced projection method* [28] [29] [30]. However, the *VGraph Algorithm* has serious drawbacks [29] when the obstacles are three-dimensional:

- shortest paths do not typically traverse the vertices of the *Grown Space Obstacles*,
- there may be no paths via vertices, within the enclosing polyhedral region R , although other types of safe paths within R may exist.

Lozano-Pérez and Wesley [28] try to alleviate the drawback by introducing some additional vertices in the *VGraph* along the edges of the *Grown Space Obstacles*. However, it is unclear how many nodes should be added in the *VGraph* to get a good approximation to the shortest path in 3D. The number of additional nodes will increase the memory space and the complexity of the *VGraph*, which will result in an enormous increase of graph search time. Therefore, the better approximation to the shortest path in 3D is needed but without increasing the complexity of the *VGraph*. The *Branch and Bound Method* [27] [38] in nonlinear programming could be an alternative that does not increase the complexity of the *VGraph*. However, it needs long compu-

tational time because of its numerical approach and it gives only some boundaries of each node for an approximation to the shortest path after long computational time. So, the *Recursive Compensation Algorithm* is proposed in order to guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph* and the better approximation to the shortest path in 3D. Therefore, a new algorithm, called the *Extended VGraph Algorithm*, should deal with the drawbacks of the *VGraph Algorithm*.

The *Extended VGraph Algorithm* has the following design steps:

- 1) Apply the *Orthogonal Projection Method* to get the *Grown Space Obstacles* in 3D.
 - i) Project obstacles in 3D onto the projection spaces.
 - ii) Build the *Grown Space Obstacles* in 2D.
 - iii) Select the necessary *Grown Space Obstacles* for the *VGraph*.
 - iv) Reconstruct the *Grown Space Obstacles* in 3D.
- 2) Find the visible vertices by detecting interferences.
- 3) Build the *VGraph* with a set of the visible vertices.
- 4) Search the *VGraph* by the graph search algorithm.
- 5) Apply the *Recursive Compensation Algorithm* to obtain the collision-free shortest path in 3D.

3.2 Preliminary Results

- The *VGraph Algorithm* has been implemented to find the collision-free shortest path in two dimensional space. This *VGraph Algorithm* can deal with not only translations of a moving object but also its rotations by using the θ sliced projection. For the simulation of the *VGraph Algorithm*, see the Problem Statement 3.2.1 and the Problem Statement 3.2.2.
- The *Orthogonal Projection Method* has been implemented to build the *Grown Space Obstacles* in 3D and to represent them in three projected two-dimensional spaces. Since the *Orthogonal Projection Method* avoids building the unnecessary *Grown Space Obstacles*, it can make the *VGraph* simpler than any other algorithms that use all of the *Grown Space Obstacles*. Therefore, the *Orthogonal Projection Method* can save the memory space to store the representation of the *Grown Space Obstacles* and it can shorten the graph search time because of the simpler *VGraph*. For the simulation

of the *Orthogonal Projection Method*, see the Problem Statement 3.2.3.

- The *Recursive Compensation Algorithm* has been implemented to find the collision-free shortest path in 3D. The *Recursive Compensation Algorithm* can guarantee the convergence to the shortest path in 3D without increasing the complexity of the *VGraph*. The property of convergency of the *Recursive Compensation Algorithm* is proved by the Theorem 2.7.4. Since ϵ is set to 10^{-5} , Lozano-Pérez's alleviation method needs a lot of memory space to store $(2 + 8 \times n \times \epsilon^{-1})$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2 + 8 \times n)$ vertices for the *VGraph*. The accuracy is defined by ϵ whose value is very small and n is the number of obstacles in workspace. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time. For the simulation of the *Recursive Compensation Algorithm*, see the Problem Statement 3.2.4.
- The *Extended VGraph Algorithm* has been presented to solve the drawbacks of the *VGraph Algorithm*. Each module of the *Extended VGraph Algorithm* has been implemented in the Problem Statement 3.2.2, the Problem Statement 3.2.3 and the Problem Statement 3.2.4.

[Problem Statement 3.2.1] Consider the problem, shown in Fig. 2.2.1, assuming that the objects are polyhedrons and their visual informations are available and they are represented by vertices. Find the collision-free shortest distance from *Start* to *Goal* with $\frac{\pi}{4}$ sliced rotation.

Table 3.2.1 Simulation result of the *VGraph* algorithm.

The shortest path is calculated by the <i>VGraph</i> Algorithm.			
Start Node = 1, Goal Node = 27,			
Path represented by internal nodes: 1 \longrightarrow 16 \longrightarrow 6 \longrightarrow 27			
From	To	Cost	Rotation
Start in 0 sliced	A_3 in $\frac{\pi}{2}$ sliced	4.125	$\frac{\pi}{2}$
A_3 in $\frac{\pi}{2}$ sliced	B_1 in $\frac{\pi}{2}$ sliced	5.025	0
B_1 in $\frac{\pi}{2}$ sliced	Goal in 0 sliced	4.031	$-\frac{\pi}{2}$
The total cost between Start and Goal = 13.179			

Fig. 2.4.2 shows the collision-free shortest path with $\frac{\pi}{4}$ sliced rotation by the *VGraph Algorithm*. The path with $\frac{\pi}{4}$ sliced rotation has 13.179 *Euclidean* distance, while the path without sliced rotation has 25.452 *Euclidean* distance. The path segment with sliced rotation is described in the Table 3.2.1, the path segment without sliced rotation

is $\{Start \rightarrow C_1 \rightarrow B_3 \rightarrow B_2 \rightarrow Goal\}$. Hence, the sliced rotation of the moving object can shorten the *Euclidean* distance. However, there is a trade off between accuracy and speed. If the small sliced rotation is considered, then the better approximation to the shortest path can be obtained, but more memory space to store each *VGraph* is needed. The result for the Problem Statement 3.2.1 comes from the file [PATH] in Appendix B. The programming list of this simulation is available in the Appendix A.

[Problem Statement 3.2.2] Assuming that the horizontal length of the moving object is 2, its vertical length is 1 and θ is $\frac{\pi}{8}$ and obstacles are given as in Fig. 2.3.5, draw the rotational *GSpace Obstacles*.

Fig. 2.3.6 - Fig. 2.3.11 draw the rotational *GSpace Obstacles*. The Problem Statement 3.2.2 shows that the *VGraph Algorithm* can handle the rotation of a moving object by the θ sliced projection method. The result of the Problem Statement 3.2.2 comes from the file [ROTATION] in Appendix D. The programming list for this simulation is available in Appendix C.

[Problem Statement 3.2.3] Build the *Grown Space Obstacles* in 3D by using the *Orthogonal Projection Method*, assuming that the object in 3D is a polyhedron, shown in Fig. 2.6.1. The object has the following vertices; $P_1(x_1, y_1, z_1)$, $P_2(x_1, y_2, z_1)$, $P_3(x_2, y_2, z_1)$, $P_4(x_2, y_1, z_1)$, $P_5(x_1, y_1, z_2)$, $P_6(x_1, y_2, z_2)$, $P_7(x_2, y_2, z_2)$, $P_8(x_2, y_1, z_2)$, where $x_1 = 7, y_1 = 5, z_1 = 3, x_2 = 14, y_2 = 10, z_2 = 12$.

Fig. 2.6.2 describes three Orthogonal Projections of Workspace D. Fig. 2.6.3 describes the *Grown Space Obstacles* in 2D. Fig. 2.6.4 describes the reconstruction of the *Grown Space Obstacles* in 3D. Fig. 2.6.5 describes the *Grown Space Obstacles* of Workspace D. The result for the Problem Statement 3.2.3 comes from the file [PROJECTION] in Appendix J. The programming list for this simulation is available in Appendix I.

[Problem Statement 3.2.4] Suppose the following vertices are calculated by the *VGraph Algorithm*; $S(3, 2, 4)$, $N_1(7, 4, 10)$, $N_2(8, 8, 9)$, $G(4, 11, 2)$ shown in Fig. 2.7.1. Calculate the shortest path from the S node to the G node, assuming that the obstacles are polyhedrals.

When ϵ is set to 10^{-5} , the *Branch and Bound Method* needs 2301666 milliseconds, however, the *RCA* needs only 416 milliseconds on VAX 11/750. The *Euclidean* distance obtained by the *RCA* is 48% less than that obtained by the *VGraph Algorithm*. The *RCA* is 55,000 times faster than the *Branch and Bound Method* within the same ϵ from the Table 3.2.4. Fig. 2.7.5 describes how fast the *RCA* works. It is proved that the sequences generated by the *RCA* are *Cauchy sequences* by the

Theorem 2.7.4. Therefore, the number of recursive compensation could be calculated if ϵ is known. Or ϵ could be calculated if the number of recursive compensation is given. Let's compare the Lozano-Pérez's alleviation method and the *Recursive Compensation Algorithm* to get the same accuracy, ϵ , for the Problem Statement 3.2.4. Since ϵ is set to 10^{-5} , Lozano-Pérez's alleviation method needs a lot of memory space to store $(2 + 2 \times 8 \times 10^5)$ vertices for the *VGraph*, while the *Recursive Compensation Algorithm* needs small memory space to store $(2 + 2 \times 8)$ vertices for the *VGraph*. Simplifying the *VGraph*, the *Recursive Compensation Algorithm* can save not only the memory space but also the graph search time. The result of the Problem Statement 3.2.4 comes from the file *[BB_{output}]* in Appendix F and the file *[RCA_{output}]* in Appendix H. The programming list of this simulation is available in the Appendix E and Appendix G.

Table. 3.2.4 Euclidean distance and Computing time

Algorithm	Distance	Computing time
VGraph	20.3283	0
Branch and Bound	13.7416	2.301.666
RCA	13.7416	416

3.3 Proposed Work

- Improve the θ sliced projection method by an algorithm to select the proper θ .
- Simplify the *VGraph* of the polygon by the *Convex Rope Algorithm*.
- Solve the collision-free Findpath problem for the dynamic obstacles.
- Exchange the knowledge on the *path planning* with other coordinators for the intelligent robot control.
- Compare the *Extended VGraph Algorithm* with other algorithms.

REFERENCES

1. Aho, A. V. and Ullman, J. D., "Data Structures and Algorithms," *Addison-Wesley*, Reading, Mass., 1983.
2. Akman, Varol, "Shortest Paths Avoiding Polyhedral Obstacles in 3-Dimensional Euclidean Space," *Ph.D. dissertation*, Dep. Computer and System Eng., Rensselaer Polytechnic Institute, 1985.
3. Aldefeld, B., "On Automatic Recognition of 3D Structures from 2D Representation," *Computer Aided Design*, Vol. 15, No. 2, March 1980, pp. 59-64.
4. Ahuja, N. and Chien, R. T. and Yen, R. and Bridwel, N., "Interference Detection and Collision Avoidance among Three Dimensional Objects," *1st Ann. Nat. Conf. Artificial Intelligence*, Stanford Univ., Stanford, CA, August 1980, pp. 44-48.
5. Bajaj, Chanderjit, "An Efficient Parellel Solution for Euclidean Shortest Path in Three Dimensions," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1897-1900.
6. Barte, Robert G. and Sherbert, Donald R., "Introduction to Real Analysis," *John Wiley* 1982.
7. Boyse, J. W., "Interference Detection among Solids and Surfaces," *Communications of the ACM*, Vol. 22, No. 1, January 1979, pp. 3-9.
8. Brooks, Rodney A., "Solving the Find-Path Problem by Good Representation of Free Space," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-13, No. 3, March/April 1983, pp. 190-197.
9. Brooks, Rodney A., "Planning Collision-Free Motions for Pick-and-Place Operations," *The International Journal of Robotics Research*, Vol. 2, No. 4, Winter 1983, pp. 19-44.
10. Brooks, Rodney A. and Lozano-Perez, Tomas, "A Subdivision Algorithm in Configuration Space for Find-Path with Rotation," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-15, No. 2, March/April 1985, pp. 224-233.
11. Chung, C. H. and Saridis, G. N., "An Obstacle Avoidance Motion Organizer for an Intelligent Robot," *Technical Report RAL-TR-88-117*, Robotics and Automation Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
12. Clocksin, W. F. and Mellish, C. S., "Programming in Prolog," *Springer-Verlag*, New York, 1987.
13. Davis, R. H. and Camacho, M., "The Application of Logic Programming to the Generation of Paths for Robots," *Robotica*, Vol.

- 2, 1984, pp. 93-103.
14. Dupont, Pierre E. and Derby, Stephen, "Planning Collision Free Paths for Redundant Robots Using a Selective Search of Configuration Space," *ASME Mechanisms Conference 86*, 1986.
15. Dupont, Pierre E., "Planning Collision Free Paths for Kinematically Redundant Robots by Selectively Mapping Configuration Space," *Ph.D dissertation*, Dep. Mech. Eng., Rensselaer Polytechnical Institute, 1988.
16. Faverjon, Bernard, "Obstacle Avoidance Using an Octree in the Configuration Space of a Manipulator," in *Proc. IEEE Int. Conf. Robotics*, Atlanta, GA, March 1984, pp. 504-512.
17. Fu, K. S. and Gonzalez, R. C. and Lee, C. S. G., "Robotics: Control, Sensing, Vision and Intelligence," *McGraw-Hill Book Company*, 1987.
18. Gilbert, Elmer G. and Johnson, Daniel W., "The Application of Distance Functions to the Optimization of Robot Motion in the Presence of Obstacles," in *Proc. 23rd Conf. Decision and Control*, December 1984, pp. 1338-1344.
19. Gilbert, Elmer G., "Distance Functions and Their Application to Robot Path Planning in the Presence of Obstacles," *IEEE Journal of Robotics and Automation*, Vol. RA-1, No. 1, March 1985, pp. 21-30.
20. Gilbert, Elmer G. and Johnson, D. W. and Keerthi, S. S., "A Fast Procedure for Computing the Distance between Complex Objects in Three Dimensional Space," *IEEE Journal of Robotics and Automation*, Vol. 4, No. 2, April 1988, pp. 193-203.
21. Gouzenes, Laurent, "Strategies for Solving Collision Free Trajectories Problems for Mobile and Manipulator Robots," *The International Journal of Robotics Research*, Vol. 3, No. 4, Winter 1984, pp. 51-65.
22. Hart, P. and Nilsson, N. J. and Raphael, B. A., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybernetics*, Vol. SSC-4, No. 2, July 1980, pp. 100-107.
23. Hayward, Vincent, "Fast Collision Detection Scheme by Recursive Decomposition of a Manipulator Wrokspace," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1044-1049.
24. Herman, Martin, "Fast, Three-Dimensional, Collision Free Motion Planning," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1056-1063.
25. Hunter, Gregory M. and Steiglitz, Kenneth, "Operations on Images Using Quad Trees," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-1, No. 2, April 1979, pp. 145-153.
26. Khatib, Oussama, "Real-Time Obstacle Avoidance for Manipu-

- lators and Mobile Robots," *The International Journal of Robotics Research*, Vol. 5, No. 1, Spring 1986, pp. 90-98.
27. Lawler, E. L. and Wood D. E., "Branch and Bound Methods: A Survey," *Operations Research*, July-August 1966, pp. 699-719.
 28. Lozano-Pérez Tomás and Wesley, Michael A., "An Algorithm for Planning Collision-Free Paths among Polyhedral Obstacles," *Communications of the ACM*, Vol. 22, No. 10, October 1979, pp. 560-570.
 29. Lozano-Pérez Tomás, "Automatic Planning of Manipulator Transfer Movements," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC-11, No. 10, October 1981, pp. 681-698.
 30. Lozano-Pérez Tomás, "Spatial Planning: A Configuration Space Approach," *IEEE Trans. Computers*, Vol. C-32, No. 2, February 1983, pp. 108-120.
 31. Lozano-Pérez Tomás and Jones, Joseph L. and Mazer, Emmanuel and O'Donnell, Patrick A. and Grimson, L., "Handy: A Robot System that Recognizes, Plans, and Manipulates," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1987, pp. 843-849.
 32. Lozano-Pérez Tomás, "A Simple Motion-Planning Algorithm for General Robot Manipulators," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, June 1987, pp. 224-238.
 33. Luh, J. Y. S. and Campbell, C. E., "Collision-Free Path Planning for Industrial Robots," in *Proc. 21th IEEE Conf. Decision and Control*, 1982, pp. 84-88.
 34. Lumelsky, Vladimir J., "Continuous Motion Planning in Unknown Environment for a 3D Cartesian Robot Arm," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1986, pp. 1050-1055.
 35. Lumelsky, Vladimir J., "Effect of Kinematics on Motion Planning for Planar Robot Arms Moving amidst Unknown Obstacles," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, June 1987, pp. 207-223.
 36. Luo, G. L. and Saridis, George N., "Optimal/PID Formulation for Control of Robotic Manipulators," *Tech. Report RAL-TR-84-034*, Robotics and Automation Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
 37. Luo, G. L. and Saridis, George N., "L-Q Design of PID Controllers for Robot Arms," *IEEE Journal of Robotics and Automation*, Vol. RA-1, No. 3, Sep. 1985, pp. 152-159.
 38. Mangasarian, O. L. and Meyer, R. R. and Robinson, S. M., "Nonlinear Programming 2," *The Academic Press*, New York, 1975.
 39. Maron, Melvin J., "Numerical Analysis," *Macmillan Publishing Co., Inc.* 1982.
 40. Martelli, A. and Motanari, U., "From Dynamic Programming to

- Search Algorithms with Functional Costs," *Proc. 4th International Joint Conference on Artificial Intelligence*, 1975, pp. 345-350.
41. Martelli, A., "On the Complexity of Admissible Search Algorithms," *Artificial Intelligence*, 1977, pp. 1-13.
 42. Maruyama, K., "A Procedure to determine intersections between polyhedral objects," *Int. J. of Computer and Information Sciences*, 1972, pp. 255-266.
 43. Meagher, Donald, "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer," *Technical Report IPL-TR-80-111*, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
 44. Meagher, Donald, "Octree Generation, Analysis and Manipulation," *Technical Report IPL-TR-82-027*, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
 45. Meyer, Walter and Benedict, Powell, "Path Planning and the Geometry of Joint Space Obstacles," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1988, pp. 215-219.
 46. Nilsson, N., "A Mobile Automation: An Application of Artificial Intelligence Technique," in *Proc. 1st International Joint Conference on Artificial Intelligence*, 1969, pp. 509-520.
 47. Nilsson, Nils J., "Principle of Artificial Intelligence," *Tioga Publishing Company*, Palo Alto, CA, 1980.
 48. Noborio, Hiroshi and Fukuda, Shozo and Arimoto, Suguru, "Construction of the Octree Approximating Three-Dimensional Objects by Using Multiple Views," *IEEE Trans. on PAMI*, Vol. 10, No. 6, November 1988, pp. 769-782.
 49. Paul, R. P., "Manipulator Cartesian Path Control," *IEEE Trans. Syst., Man, Cybern.*, Vol. SMC-9, Nov. 1979, pp. 702-711.
 50. Paul, R. P., "Robot Manipulators: Mathematics, Programming, and Control," *The MIT Press*, Cambridge, Massachusetts, 1982.
 51. Paul, R. P. and Zhang, H., "The Dynamics of the PUMA Manipulator," in *Proc. 1983 American Control Conf.*, pp. 491-496.
 52. Pearl, Judea, "Heuristics: Intelligent Search Strategies for Computer Problem Solving," *Addison-Wesley Publishing Company*, 1984.
 53. Peshkin, Michael A. and Sanderson, Arthur C., "Reachable Grasp on a Polygon: The Convex Rope Algorithm," *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, March 1986, pp. 53-58.
 54. Ray, William O., "Real Analysis," *Prentice Hall*, 1988.
 55. Saridis, George N., "Intelligent Robotic Control," *IEEE Trans. Au-*

- tomatic Control*, Vol. AC-28, No. 5, May 1983, pp. 547-557.
56. Saridis, George N. and Graham, J. H., "Linguistic Decision Schemata for Intelligent Robots," *Automatica*, Vol. 20, No. 1, 1984, pp. 121-126.
 57. Sedgewick, Robert, "Algorithms," *Addison-Wesley Publishing Company*, 1983.
 58. Tarn, T. J. and Bejczy, A. K. and Han, Shuotiao and Yun, Xiaoping, "Inertia Parameters of PUMA 560 Robot Arm," *Technical Report SSM-RL-85-01*, Robotics Laboratory, Department of Systems Science and Mathematics, Washington University, St. Louis, Missouri, 63130.
 59. Udupa, S. M., "Collision Detection and Avoidance in Computer Controlled Manipulators," in *Proc. 5th International Joint Conference on Artificial Intelligence*, MIT, Cambridge, MA, August 1977, pp. 737-748.
 60. Valavanis, Kimon P. and Saridis, George N., "A Mathematical Formulation for the Analytical Design of Intelligent Machines," *Technical Report RAL-TR-86-085*, Robotics and Automation Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 12180-3590.
 61. Valenti, Joan Mari i, "Study of New Heuristics to Compute Collision Free Paths of Rigid Bodies in a 2D Universe," *Ph.D. dissertation*, Dep. D'enginyeria Cibernètica, Universitat Politècnica De Catalunya, 1987.
 62. Winston, P. A., "Artificial Intelligence," *Addison-Wesley Publishing Company*, 1984.
 63. Wong, E. K. and Fu, K. S., "A Hierarchical Orthogonal Space Approach to Three Dimensional Path Planning," *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, March 1986, pp. 42-53.

Appendix A: Simulation of the VGraph Algorithm

```
1 program VGA(OBSTACLES,GSPACE,VERTICES,PATH,output);
2 {
3
4   Author   :   C. H. Chung
5
6   Version  :   4.7
7
8   Date     :   December 1, 1988
9
10
11
12   This program is designed to simulate the VGraph Algorithm.
13
14   It consists mainly of 3 procedures.
15
16   1. BuildGSpaceObstacles(List).
17       . To build the Grown Space Obstacles
18       . INPUT FILE : OBSTACLES, (GSPACE)
19       . OUTPUT FILE : output, GSPACE
20       . output of this procedure : List
21
22   2. BuildVGraph(A, List).
23       . To build the VGraph
24       . The first part of this procedure mainly consists
25       . of the Interference Checking, i.e. the Visible
26       . Vertices, and the second part of this procedure
27       . mainly consists of the VGraph Construction.
28       . INPUT FILE : _____
29       . OUTPUT FILE : VERTICE
30       . input of this procedure : List
31       . output of this procedure : A
32
33   3. SearchVGraph(A, LinkedPath).
34       . To find the shortest path of the VGraph
35       . LinkedPath holds the information of the shortest
36       . path by the VGraph Algorithm.
37       . INPUT FILE : _____
38       . OUTPUT FILE : PATH
39       . input of this procedure : A
40       . output of this procedure : LinkedPath
41
42
43
44   Pay a special attentation on the data structure of List.
45   List consists of S1, S2,
46       A1(1), A2(1), B1(1), B2(1), C1(1), C2(1),
47       A1(2), A2(2), B1(2), B2(2), C1(2), C2(2),
48       A1(3), A2(3), B1(3), B2(3), C1(3), C2(3),
49       A1(4), A2(4), B1(4), B2(4), C1(4), C2(4),
50       G1, G2.
```



```

51
52
53 type
54     PointType = record
55         x, y : real;
56     end;
57     VerticeType = ^Nodes;
58     Nodes = record
59         Node : PointType;
60         Next : VerticeType;
61     end;
62     PathType = ^Item;
63     Item = record
64         Data : integer;
65         Next : PathType;
66     end;
67     CostMatrix = array [1..28,1..28] of real;
68
69 var
70     A : CostMatrix;
71     List : VerticeType;
72     LinkedPath : PathType;
73     OBSTACLES, GSPACE, VERTICES, PATH : text;
74
75
76
77
78
79
80
81 procedure BuildGSpaceObstacles(var List : VerticeType);
82 {
83
84     Author   :   C. H. Chung
85
86     Version  :   2.3
87
88     Date     :   November 29, 1988
89
90
91
92     Procedure BuildGSpaceObstacles(List) .
93         . To build the Grown Space Obstacles
94         . INPUT  FILE : OBSTACLES, (GSPACE)
95         . OUTPUT FILE : output, GSPACE
96         . output of this procedure : List
97
98
99     This program will build the Grown Space Obstacles.
100

```

```

101      hh : the horizontal length of the object
102
103      vv : the vertical length of the object
104
105      rr : the sliced angle for rotational Grown Space Obstacles (radian)
106
107
108  const
109      Pi = 3.141592;      (Radian)
110  var
111      Object : VerticeType;
112      ObjectA, ObjectB, ObjectC : VerticeType;
113      hh, vv, rr : real;
114      From, To : PointType;
115
116
117
118
119
120  procedure PrintVertice(List : VerticeType);
121  (
122      This procedure will print the Linked List for INPUT.
123      List contains the Start, Goal, and Obstacles.
124
125  var
126      Current : VerticeType;
127  begin
128      Current := List;
129      writeln;
130      writeln(Current^.Node.x :10:3, Current^.Node.y :10:3);
131      Current := Current^.Next;
132      while (Current<> nil)
133      do begin
134          writeln(Current^.Node.x :10:3, Current^.Node.y :10:3);
135          Current := Current^.Next;
136      end;
137      writeln;
138  end;
139
140
141
142
143
144
145
146  procedure CreateObject(var Object : VerticeType);
147  (
148      This procedure creates the object from the input file
149      by the linked list.
150

```

```

151     var
152         Current : VerticeType;
153     begin
154         Object := nil;
155         if not eof(OBSTACLES)
156             then begin
157                 new(Object);
158                 readln(OBSTACLES, Object^.Node.x,
159                     Object^.Node.y);
160                 Object^.Next := nil;
161                 Current := Object;
162                 while not eof(OBSTACLES)
163                     do begin
164                         new(Current^.Next);
165                         Current := Current^.Next;
166                         readln(OBSTACLES, Current^.Node.x,
167                             Current^.Node.y);
168                         Current^.Next := nil
169                     end
170             end
171     end;
172
173
174
175
176
177
178 procedure CreateList(var List : VerticeType; From, To : PointType)
179 {
180     This procedure creates the object from the input file
181     by the linked list.
182 }
183 var
184     Current : VerticeType;
185     Flag : boolean;
186 begin
187     List := nil;
188     if not eof(GSPACE)
189         then begin
190             new(List);
191             List^.Node := From;
192             List^.Next := nil;
193             Current := List;
194
195             new(Current^.Next);
196             Current := Current^.Next;
197             Current^.Node := From;
198             Current^.Next := nil;
199
200             Flag := true;

```

```

201         while Flag and (not eof(GSPACE))
202             do begin
203                 new(Current^.Next);
204                 Current := Current^.Next;
205                 readln(GSPACE, Current^.Node.x,
206                     Current^.Node.y);
207                 Current^.Next := nil;
208                 if ((Current^.Node.x = 7.0) and (* bad *)
209                     (Current^.Node.y = 1.0)) {STOPPING}
210                     then Flag := false;      { * CASE *
211             end;
212
213             new(Current^.Next);
214             Current := Current^.Next;
215             Current^.Node := To;
216             Current^.Next := nil;
217
218             new(Current^.Next);
219             Current := Current^.Next;
220             Current^.Node := To;
221             Current^.Next := nil;
222         end;
223     end;
224
225
226
227
228
229
230
231 procedure GrownObject(var Object, Grown : VerticeType;
232                     hh, vv, rr : real);
233 {
234     This procedure builds the Grown Space Obstacles.
235
236     where h : horizontal length
237           v : vertical length
238
239     0 < q < Pi/2
240
241         a1 = (A1x,A1y) + h(-cos(q),-sin(q))
242         a2 = (A1x,A1y)
243         a3 = (A2x,A2y)
244         a4 = (A2x,A2y) + v(sin(q),-cos(q))
245         a5 = (A3x,A3y) + v(sin(q),-cos(q))
246         a6 = (a5x,a5y) + h(-cos(q),-sin(q))
247         a8 = (A4x,A4y) + h(-cos(q),-sin(q))
248         a7 = (a8x,a8y) + v(sin(q),-cos(q))
249
250     Pi/2 < q < Pi

```

```

251
252         q = q - Pi/2
253         temp = h      (to swap h and v)
254         h = v
255         v = temp
256
257     q = 0
258         Delete a2, a4, a6, a8.
259
260     q = Pi/2
261         Swap h and v.
262         Delete a2, a4, a6, a8.
263
264 var
265     Current, Head : VerticeType;
266 begin
267     Current := nil;
268     new(Current);
269     Current^.Node.x := Object^.Node.x - hh * cos(rr);
270     Current^.Node.y := Object^.Node.y - hh * sin(rr);
271     Current^.Next := nil;
272
273     Head := Object;
274     Grown := Current;
275
276     new(Current^.Next);
277     Current := Current^.Next;
278     Current^.Node.x := Object^.Node.x;
279     Current^.Node.y := Object^.Node.y;
280     Current^.Next := nil;
281
282     Object := Object^.Next;
283     new(Current^.Next);
284     Current := Current^.Next;
285     Current^.Node.x := Object^.Node.x;
286     Current^.Node.y := Object^.Node.y;
287     Current^.Next := nil;
288
289     new(Current^.Next);
290     Current := Current^.Next;
291     Current^.Node.x := Object^.Node.x + vv * sin(rr);
292     Current^.Node.y := Object^.Node.y - vv * cos(rr);
293     Current^.Next := nil;
294
295     Object := Object^.Next;
296     new(Current^.Next);
297     Current := Current^.Next;
298     Current^.Node.x := Object^.Node.x + vv * sin(rr);
299     Current^.Node.y := Object^.Node.y - vv * cos(rr);
300     Current^.Next := nil;

```

```

301
302     new(Current^.Next);
303     Current := Current^.Next;
304     Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr);
305     Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr);
306     Current^.Next := nil;
307
308     Object := Object^.Next;
309     new(Current^.Next);
310     Current := Current^.Next;
311     Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr);
312     Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr);
313     Current^.Next := nil;
314
315     new(Current^.Next);
316     Current := Current^.Next;
317     Current^.Node.x := Object^.Node.x - hh * cos(rr);
318     Current^.Node.y := Object^.Node.y - hh * sin(rr);
319     Current^.Next := nil;
320
321     Object := Head;
322 end;
323
324
325
326
327
328
329
330 procedure RotationalGrowth(ObjectA, ObjectB, ObjectC : VerticeType;
331                             rr, hh, vv : real);
332
333     (
334         This procedure will print the Grown Space Obstacles, considering
335         the rotational effect.
336     )
337
338 var
339     GrownA1, GrownA2,
340     GrownB1, GrownB2,
341     GrownC1, GrownC2 : VerticeType;
342
343 begin
344     GrownObject(ObjectA, GrownA1, hh, vv, rr*0);
345     GrownObject(ObjectB, GrownB1, hh, vv, rr*0);
346     GrownObject(ObjectC, GrownC1, hh, vv, rr*0);
347     GrownObject(ObjectA, GrownA2, vv, hh, rr*2 - Pi/2);
348     GrownObject(ObjectB, GrownB2, vv, hh, rr*2 - Pi/2);
349     GrownObject(ObjectC, GrownC2, vv, hh, rr*2 - Pi/2);
350
351     while (GrownA1 <> nil) do
352         begin

```

```

351         writeln(GSPACE, GrownA1^.Node.x :10:4,
352                 GrownA1^.Node.y :10:4);
353         writeln(GSPACE, GrownA2^.Node.x :10:4,
354                 GrownA2^.Node.y :10:4);
355         writeln(GSPACE, GrownB1^.Node.x :10:4,
356                 GrownB1^.Node.y :10:4);
357         writeln(GSPACE, GrownB2^.Node.x :10:4,
358                 GrownB2^.Node.y :10:4);
359         writeln(GSPACE, GrownC1^.Node.x :10:4,
360                 GrownC1^.Node.y :10:4);
361         writeln(GSPACE, GrownC2^.Node.x :10:4,
362                 GrownC2^.Node.y :10:4);
363
364         GrownA1 := GrownA1^.Next^.Next;
365         GrownA2 := GrownA2^.Next^.Next;
366         GrownB1 := GrownB1^.Next^.Next;
367         GrownB2 := GrownB2^.Next^.Next;
368         GrownC1 := GrownC1^.Next^.Next;
369         GrownC2 := GrownC2^.Next^.Next;
370         writeln(GSPACE);
371     end;
372 end; { of Procedure RotationalGrowth }
373
374
375
376
377
378
379
380 procedure Partition(var Object, ObjectA, ObjectB,
381                   ObjectC : VerticeType);
382 (
383     This procedure will partition the whole Object into 3 small
384     objects (ObjectA, ObjectB, ObjectC).
385 )
386 var
387     i : integer;
388     Current, CurrentA, CurrentB, CurrentC : VerticeType;
389 begin
390     Current := Object;
391
392     ObjectA := nil;
393     new(ObjectA);
394     ObjectA^.Node := Current^.Node;
395     ObjectA^.Next := nil;
396     CurrentA := ObjectA;
397     for i := 1 to 3 do
398     begin
399         new(CurrentA^.Next);
400         Current := Current^.Next;

```

```

401         CurrentA := CurrentA^.Next;
402         CurrentA^.Node := Current^.Node;
403         CurrentA^.Next := nil;
404     end;
405
406     Current := Current^.Next;
407
408     ObjectB := nil;
409     new(ObjectB);
410     ObjectB^.Node := Current^.Node;
411     ObjectB^.Next := nil;
412     CurrentB := ObjectB;
413     for i := 1 to 3 do
414     begin
415         new(CurrentB^.Next);
416         Current := Current^.Next;
417         CurrentB := CurrentB^.Next;
418         CurrentB^.Node := Current^.Node;
419         CurrentB^.Next := nil;
420     end;
421
422     Current := Current^.Next;
423
424     ObjectC := nil;
425     new(ObjectC);
426     ObjectC^.Node := Current^.Node;
427     ObjectC^.Next := nil;
428     CurrentC := ObjectC;
429     for i := 1 to 3 do
430     begin
431         new(CurrentC^.Next);
432         Current := Current^.Next;
433         CurrentC := CurrentC^.Next;
434         CurrentC^.Node := Current^.Node;
435         CurrentC^.Next := nil;
436     end;
437 end; { of procedure Partition }
438
439
440
441
442 begin { _____ BuildGSpaceObstacles _____ }
443
444     reset(OBSTACLES);
445     rewrite(GSPACE);
446
447     rr := Pi / 4;
448     hh := 2;
449     vv := 1;
450

```



```

451      CreateObject(Object);
452      Partition(Object, ObjectA, ObjectB, ObjectC);
453      RotationalGrowth(ObjectA, ObjectB, ObjectC, rr, hh, vv);
454
455      reset(GSPACE); (*****)
456
457      From.x := 5.0;          ( From, To )
458      From.y := 7.0;
459
460      To.x := 13.0; (*****);
461      To.y := 15.0;
462
463      CreateList(List, From, To);
464
465      writeln(' ____ Linked List for INPUT ____ ');
466      PrintVertice(List);
467      writeln;
468      writeln;
469      writeln(' INPUT FILE : OBSTACLES');
470      writeln;
471      writeln(' OUTPUT FILE : GSPACE (for the Grown Space Obstacles
472      writeln(' VERTICES (for the Visible Vertices)');
473      writeln(' PATH (for the shortest path)');
474      writeln(' output (for this display)');
475
476      end; ( of Procedure BuildGSpaceObstacles )
477
478
479
480
481
482
483
484 procedure BuildVGraph(var A : CostMatrix; var List : VerticeType);
485 (
486
487     Author : C. H. Chung
488
489     Version : 2.0
490
491     Date : November 17, 1988
492
493
494
495     This program will build the Visibility Graph.
496
497     BuildVGraph(A, List)
498         . To build the VGraph
499         . The first part of this procedure mainly consists of
500           the Interference Checking, i.e. the Visible Vertices,

```

```

501         and the second part of this procedure mainly consists
502         of the VGraph Construction.
503
504     Pay a special attention on the data structure of List.
505     List consists of S1, S2,
506             A1(1), A2(1), B1(1), B2(1), C1(1), C2(1),
507             A1(2), A2(2), B1(2), B2(2), C1(2), C2(2),
508             A1(3), A2(3), B1(3), B2(3), C1(3), C2(3),
509             A1(4), A2(4), B1(4), B2(4), C1(4), C2(4),
510             G1, G2.
511
512 }
513 var
514     From, To : VerticeType;
515     i, j, n : integer;
516
517
518
519
520
521
522
523 {
524     This procedure is very useful in printing a Linked List
525     to debug the procedure of BuildVGraph. So, this procedure
526     will remain in the main programming sheet for the future
527     debugging.
528
529
530     procedure PrintVertice(List : VerticeType);
531
532         This procedure will print the Linked List of the shortest path.
533
534     var
535         Current : VerticeType;
536     begin
537         Current := List;
538         writeln(VERTICES, Current^.Node.x :10:3,
539               Current^.Node.y :10:3);
540         Current := Current^.Next;
541         while (Current<> nil)
542             do begin
543                 writeln(VERTICES, Current^.Node.x :10:3,
544                       Current^.Node.y :10:3);
545                 Current := Current^.Next;
546             end;
547         writeln(VERTICES);
548     end;
549
550

```

```

551
552
553
554
555
556
557 procedure PrintAMatrix(A : CostMatrix);

```

```

558 (
559     This procedure will print a Matrix.
560 )

```

```

561 var
562     i, j : integer;
563
564 begin
565     for i := 1 to 28 do
566         begin
567             writeln(VERTICES);
568             for j := 1 to 28 do
569                 if A[i,j] < 999
570                     then writeln(VERTICES, '  A[' , i :2:0, ', ',
571                                     j :2:0, ']' = ', A[i,j] :10:4);
572             end;
573         end;
574     end;
575
576
577
578
579
580

```

```

581 procedure LineEquation(From, To : PointType; var a, b : real);

```

```

582 (
583     This procedure will generate a line equation through two points
584 )

```

```

585
586         Y2 - Y1
587     Y := -----(X - X1) + Y1
588         X2 - X1
589
590         Y2 - Y1      X2*Y1 - Y2*X1
591     := -----(X) + -----
592         X2 - X1      X2 - X1
593
594     := a*X + b
595

```

```

596
597 begin
598     a := (To.y - From.y) / (To.x - From.x);
599     b := (To.x * From.y - To.y * From.x) / (To.x - From.x);
600 end;

```

```

601
602
603
604
605
606
607
608 function max(a, b : real): real;
609 {
610     This function calculate the maximum.
611 }
612 begin
613     if a > b
614         then max := a
615         else max := b;
616 end;
617
618
619
620
621
622
623 function min(a, b : real): real;
624 {
625     This function will calculate the minimum.
626 }
627 begin
628     if a > b
629         then min := b
630         else min := a;
631 end;
632
633
634
635
636
637
638
639 function FinalCheck(a, b, X1, X2, X3, X4,
640                     Y1, Y2, Y3, Y4 : real): boolean;
641 {
642     This function will find the Interference in the normal case.-
643 }
644
645 var
646     X, Y,
647     Xmin, Ymin,
648     Xmax, Ymax : real;
649
650 begin

```

```

651      Xmin := max(X1, X3);
652      Xmax := min(X2, X4);
653      Ymin := max(Y1, min(Y3, Y4));
654      Ymax := min(Y2, max(Y3, Y4));
655
656      Y := Y2;
657      X := (Y - b) / a;
658      if (X > Xmin) and (X < Xmax)
659      then FinalCheck := true
660      else begin
661          X := X2;
662          Y := a * X + b;
663          if (Y > Ymin) and (Y < Ymax)
664          then FinalCheck := true
665          else begin
666              Y := Y1;
667              X := (Y - b) / a;
668              if (X > Xmin) and (X < Xmax)
669              then FinalCheck := true
670              else begin
671                  X := X1;
672                  Y := a * X + b;
673                  if (Y > Ymin) and (Y < Ymax)
674                  then FinalCheck := true
675                  else FinalCheck := false;
676              end;
677          end;
678      end;
679  end;
680
681
682
683
684
685
686
687  function DetectInterference(Object : VerticeType;
688                               From, To : PointType) : boolean;
689  (
690      This function will classify the Interference.
691  )
692
693  var
694      a, b,
695      tempX, tempY,
696      X1, X2, X3, X4,
697      Y1, Y2, Y3, Y4 : real;
698      Current : VerticeType;
699
700  begin

```

```

701      Current := Object;
702
703      {1.  Swap From and To by the X position,
704          in order to set From to the left of To.}
705      if From.x > To.x
706      then begin
707          tempX := From.x;
708          tempY := From.y;
709          From.x := To.x;
710          From.y := To.y;
711          To.x := tempX;
712          To.y := tempY;
713      end;
714
715      {2.  Define X1, X2, X3, X4, Y1, Y2, Y3, Y4.}
716      X1 := Current^.Node.x;
717      Y2 := Current^.Node.y;
718      Current := Current^.Next;
719      X2 := Current^.Node.x;
720      Current := Current^.Next;
721      Y1 := Current^.Node.y;
722      Current := Object;
723      X3 := From.x;
724      Y3 := From.y;
725      X4 := To.x;
726      Y4 := To.y;
727
728      {3.  Find the x, y boundary of the object.
729          However, Step 2 implies Step 3.}
730
731      {4.  Find the line equation throught From and To and its boundary.}
732      if (X3 = X4) and (Y3 = Y4)
733      then DetectInterference := false
734      else if (X3 = X4)
735      then if (X3 > X1) and (X3 < X2)
736      then if (min(Y3,Y4) > Y2) or (max(Y3,Y4) < Y1)
737      then DetectInterference := false
738      else DetectInterference := true
739      else DetectInterference := false
740      else if (Y3 = Y4)
741      then if (Y3 > Y1) and (Y3 < Y2)
742      then if (min(X3,X4) > X2) or
743      (max(X3,X4) < X1)
744      then DetectInterference := false
745      else DetectInterference := true
746      else DetectInterference := false
747      else begin
748          LineEquation(From, To, a, b);
749
750      {5. & 6. is implied in DetectInterference. }

```

```

751
752         if (min(Y3,Y4) >= Y2) or
753             (max(Y3,Y4) <= Y1) or
754             (min(X3,X4) >= X2) or
755             (max(X3,X4) <= X1)
756         then DetectInterference := false
757         else DetectInterference :=
758             FinalCheck(a,b,X1,X2,
759                 X3,X4,Y1,Y2,Y3,Y4)
760
761     end;
762
763
764
765
766
767
768
769
770     {
771     This procedure is very useful in checking the Interference
772     between the line and obstacles, and if there is any
773     interference, then this procedure will print out the
774     information on it. However, this procedure will remain
775     in the main programming sheet for the future debugging.
776
777     }
778     procedure PrintInformationOnInterference(Object : VerticeType;
779         From, To : PointType;
780         First, Second : char);
781
782     This procedure will print the information on Interference.
783
784     var
785         Current : VerticeType;
786     begin
787         Current := Object;
788         if DetectInterference(Current, From, To)
789         then begin
790             writeln(VERTICES, ' in Object ', First, Second);
791             PrintVertice(Current);
792         end;
793     end;
794
795
796
797
798
799
800     function Interference(List : VerticeType; From, To : PointType;

```

```

801                                COUNT : integer): boolean;
802                                {
803                                This function will find the Interference with the rotational
804                                Grown Space Obstacles.
805                                }
806
807                                var
808                                Current,
809                                HeadA1, HeadA2,
810                                HeadB1, HeadB2,
811                                HeadC1, HeadC2,
812                                ObjectA1, ObjectA2,
813                                ObjectB1, ObjectB2,
814                                ObjectC1, ObjectC2 : VerticeType;
815
816                                begin
817                                Current := List;
818                                Current := Current^.Next;           { Skip S1.}
819                                Current := Current^.Next;           { Skip S2.}
820
821                                ObjectA1 := nil;
822                                new(ObjectA1);
823                                ObjectA1^.Node := Current^.Node;
824                                ObjectA1^.Next := nil;
825                                Current := Current^.Next;
826
827                                ObjectA2 := nil;
828                                new(ObjectA2);
829                                ObjectA2^.Node := Current^.Node;
830                                ObjectA2^.Next := nil;
831                                Current := Current^.Next;
832
833                                ObjectB1 := nil;
834                                new(ObjectB1);
835                                ObjectB1^.Node := Current^.Node;
836                                ObjectB1^.Next := nil;
837                                Current := Current^.Next;
838
839                                ObjectB2 := nil;
840                                new(ObjectB2);
841                                ObjectB2^.Node := Current^.Node;
842                                ObjectB2^.Next := nil;
843                                Current := Current^.Next;
844
845                                ObjectC1 := nil;
846                                new(ObjectC1);
847                                ObjectC1^.Node := Current^.Node;
848                                ObjectC1^.Next := nil;
849                                Current := Current^.Next;
850

```



```

851 ObjectC2 := nil;
852 new(ObjectC2);
853 ObjectC2^.Node := Current^.Node;
854 ObjectC2^.Next := nil;
855 Current := Current^.Next;
856
857 HeadA1 := ObjectA1;
858 HeadA2 := ObjectA2;
859 HeadB1 := ObjectB1;
860 HeadB2 := ObjectB2;
861 HeadC1 := ObjectC1;
862 HeadC2 := ObjectC2;
863
864 while Current^.Next^.Next <> nil      ( Skip G1, G2.)
865   do begin
866     new(ObjectA1^.Next);
867     ObjectA1 := ObjectA1^.Next;
868     ObjectA1^.Node := Current^.Node;
869     ObjectA1^.Next := nil;
870     Current := Current^.Next;
871
872     new(ObjectA2^.Next);
873     ObjectA2 := ObjectA2^.Next;
874     ObjectA2^.Node := Current^.Node;
875     ObjectA2^.Next := nil;
876     Current := Current^.Next;
877
878     new(ObjectB1^.Next);
879     ObjectB1 := ObjectB1^.Next;
880     ObjectB1^.Node := Current^.Node;
881     ObjectB1^.Next := nil;
882     Current := Current^.Next;
883
884     new(ObjectB2^.Next);
885     ObjectB2 := ObjectB2^.Next;
886     ObjectB2^.Node := Current^.Node;
887     ObjectB2^.Next := nil;
888     Current := Current^.Next;
889
890     new(ObjectC1^.Next);
891     ObjectC1 := ObjectC1^.Next;
892     ObjectC1^.Node := Current^.Node;
893     ObjectC1^.Next := nil;
894     Current := Current^.Next;
895
896     new(ObjectC2^.Next);
897     ObjectC2 := ObjectC2^.Next;
898     ObjectC2^.Node := Current^.Node;
899     ObjectC2^.Next := nil;
900     Current := Current^.Next;

```

```

901
902         end;
903
904         ObjectA1 := HeadA1;
905         ObjectA2 := HeadA2;
906         ObjectB1 := HeadB1;
907         ObjectB2 := HeadB2;
908         ObjectC1 := HeadC1;
909         ObjectC2 := HeadC2;
910
911         if COUNT = COUNT div 2 * 2
912             then if DetectInterference(ObjectA2, From, To) or
913                  DetectInterference(ObjectB2, From, To) or
914                  DetectInterference(ObjectC2, From, To)
915                  then Interference := true
916                  else Interference := false
917             else if DetectInterference(ObjectA1, From, To) or
918                  DetectInterference(ObjectB1, From, To) or
919                  DetectInterference(ObjectC1, From, To)
920                  then Interference := true
921                  else Interference := false;
922
923     end;  ( Of function Interference )
924
925
926
927
928
929
930
931     function CrossDiagonal(Object : VerticeType;
932                             From, To : PointType): boolean;
933     {
934         This function will determine whether two vertices are in
935         the disgoanl of the same object.
936     }
937     var
938         Current1, Current2, Current3, Current4 : VerticeType;
939         tempX, tempY : real;
940     begin
941         Current1 := Object;
942         Current2 := Current1^.Next;
943         Current3 := Current2^.Next;
944         Current4 := Current3^.Next;
945
946         if From.x > To.x
947             then begin
948                 tempX := From.x;
949                 tempY := From.y;
950                 From.x := To.x;

```

```

951         From.y := To.y;
952         To.x := tempX;
953         To.y := tempY;
954     end;
955
956     if (Current1^.Node.x = From.x) and
957        (Current1^.Node.y = From.y) and
958        (Current3^.Node.x = To.x) and
959        (Current3^.Node.y = To.y)
960     then CrossDiagonal := true
961     else if (Current2^.Node.x = To.x) and
962            (Current2^.Node.y = To.y) and
963            (Current4^.Node.x = From.x) and
964            (Current4^.Node.y = From.y)
965     then CrossDiagonal := true
966     else CrossDiagonal := false;
967 end;    { Of function CrossDiagonal }
968
969
970
971
972
973
974
975 function CrossVertices(List : VerticeType;
976                        From, To : PointType) : boolean;
977 {
978     This function will find the Interference with the rotational
979     Grown Space Obstacles.
980 }
981
982 var
983     Current,
984     HeadA1, HeadA2,
985     HeadB1, HeadB2,
986     HeadC1, HeadC2,
987     ObjectA1, ObjectA2,
988     ObjectB1, ObjectB2,
989     ObjectC1, ObjectC2 : VerticeType;
990
991 begin
992     Current := List;
993     Current := Current^.Next;      { Skip S1.}
994     Current := Current^.Next;      { Skip S2.}
995
996     ObjectA1 := nil;
997     new(ObjectA1);
998     ObjectA1^.Node := Current^.Node;
999     ObjectA1^.Next := nil;
1000    Current := Current^.Next;

```

```

1001
1002     ObjectA2 := nil;
1003     new(ObjectA2);
1004     ObjectA2^.Node := Current^.Node;
1005     ObjectA2^.Next := nil;
1006     Current := Current^.Next;
1007
1008     ObjectB1 := nil;
1009     new(ObjectB1);
1010     ObjectB1^.Node := Current^.Node;
1011     ObjectB1^.Next := nil;
1012     Current := Current^.Next;
1013
1014     ObjectB2 := nil;
1015     new(ObjectB2);
1016     ObjectB2^.Node := Current^.Node;
1017     ObjectB2^.Next := nil;
1018     Current := Current^.Next;
1019
1020     ObjectC1 := nil;
1021     new(ObjectC1);
1022     ObjectC1^.Node := Current^.Node;
1023     ObjectC1^.Next := nil;
1024     Current := Current^.Next;
1025
1026     ObjectC2 := nil;
1027     new(ObjectC2);
1028     ObjectC2^.Node := Current^.Node;
1029     ObjectC2^.Next := nil;
1030     Current := Current^.Next;
1031
1032     HeadA1 := ObjectA1;
1033     HeadA2 := ObjectA2;
1034     HeadB1 := ObjectB1;
1035     HeadB2 := ObjectB2;
1036     HeadC1 := ObjectC1;
1037     HeadC2 := ObjectC2;
1038
1039     while Current^.Next^.Next <> nil      { Skip G1, G2.}
1040     do begin
1041         new(ObjectA1^.Next);
1042         ObjectA1 := ObjectA1^.Next;
1043         ObjectA1^.Node := Current^.Node;
1044         ObjectA1^.Next := nil;
1045         Current := Current^.Next;
1046
1047         new(ObjectA2^.Next);
1048         ObjectA2 := ObjectA2^.Next;
1049         ObjectA2^.Node := Current^.Node;
1050         ObjectA2^.Next := nil;

```

```

1051         Current := Current^.Next;
1052
1053         new(ObjectB1^.Next);
1054         ObjectB1 := ObjectB1^.Next;
1055         ObjectB1^.Node := Current^.Node;
1056         ObjectB1^.Next := nil;
1057         Current := Current^.Next;
1058
1059         new(ObjectB2^.Next);
1060         ObjectB2 := ObjectB2^.Next;
1061         ObjectB2^.Node := Current^.Node;
1062         ObjectB2^.Next := nil;
1063         Current := Current^.Next;
1064
1065         new(ObjectC1^.Next);
1066         ObjectC1 := ObjectC1^.Next;
1067         ObjectC1^.Node := Current^.Node;
1068         ObjectC1^.Next := nil;
1069         Current := Current^.Next;
1070
1071         new(ObjectC2^.Next);
1072         ObjectC2 := ObjectC2^.Next;
1073         ObjectC2^.Node := Current^.Node;
1074         ObjectC2^.Next := nil;
1075         Current := Current^.Next;
1076
1077     end;
1078
1079     ObjectA1 := HeadA1;
1080     ObjectA2 := HeadA2;
1081     ObjectB1 := HeadB1;
1082     ObjectB2 := HeadB2;
1083     ObjectC1 := HeadC1;
1084     ObjectC2 := HeadC2;
1085
1086     if CrossDiagonal(ObjectA1, From, To) or
1087        CrossDiagonal(ObjectA2, From, To) or
1088        CrossDiagonal(ObjectB1, From, To) or
1089        CrossDiagonal(ObjectB2, From, To) or
1090        CrossDiagonal(ObjectC1, From, To) or
1091        CrossDiagonal(ObjectC2, From, To)
1092     then CrossVertices := true
1093     else CrossVertices := false;
1094 end; { Of Function CrossVertices }
1095
1096
1097
1098
1099
1100

```

```

1101 function GrownDeadNode(Object : VerticeType;
1102                        Data: PointType): boolean;
1103 {
1104     This function will classify the Dead node.
1105 }
1106
1107 var
1108     X1, X2, X3,
1109     Y1, Y2, Y3 : real;
1110     Current : VerticeType;
1111
1112 begin
1113     Current := Object;
1114
1115     X1 := Current^.Node.x;
1116     Y2 := Current^.Node.y;
1117     Current := Current^.Next;
1118     X2 := Current^.Node.x;
1119     Current := Current^.Next;
1120     Y1 := Current^.Node.y;
1121     Current := Object;
1122     X3 := Data.x;
1123     Y3 := Data.y;
1124     if (X3 > X1) and (X3 < X2) and (Y3 > Y1) and (Y3 < Y2)
1125     then GrownDeadNode := true
1126     else GrownDeadNode := false;
1127 end;
1128
1129
1130
1131
1132
1133
1134
1135 function DeadNode(List : VerticeType; Data : PointType;
1136                  i : integer): boolean;
1137 {
1138     This function will find the Dead Node with the rotational
1139     Grown Space Obstacles.
1140 }
1141
1142 var
1143     Current,
1144     HeadA1, HeadA2,
1145     HeadB1, HeadB2,
1146     HeadC1, HeadC2,
1147     ObjectA1, ObjectA2,
1148     ObjectB1, ObjectB2,
1149     ObjectC1, ObjectC2 : VerticeType;
1150

```

```

- 1151      begin
1152          Current := List;
1153          Current := Current^.Next;          { Skip S1.}
1154          Current := Current^.Next;          { Skip S2.}
- 1155
1156          ObjectA1 := nil;
1157          new(ObjectA1);
- 1158          ObjectA1^.Node := Current^.Node;
1159          ObjectA1^.Next := nil;
1160          Current := Current^.Next;
- 1161
1162          ObjectA2 := nil;
1163          new(ObjectA2);
1164          ObjectA2^.Node := Current^.Node;
- 1165          ObjectA2^.Next := nil;
1166          Current := Current^.Next;
1167
- 1168          ObjectB1 := nil;
1169          new(ObjectB1);
1170          ObjectB1^.Node := Current^.Node;
1171          ObjectB1^.Next := nil;
- 1172          Current := Current^.Next;
1173
1174          ObjectB2 := nil;
- 1175          new(ObjectB2);
1176          ObjectB2^.Node := Current^.Node;
1177          ObjectB2^.Next := nil;
- 1178          Current := Current^.Next;
1179
1180          ObjectC1 := nil;
1181          new(ObjectC1);
- 1182          ObjectC1^.Node := Current^.Node;
1183          ObjectC1^.Next := nil;
1184          Current := Current^.Next;
- 1185
1186          ObjectC2 := nil;
1187          new(ObjectC2);
- 1188          ObjectC2^.Node := Current^.Node;
1189          ObjectC2^.Next := nil;
1190          Current := Current^.Next;
1191
- 1192          HeadA1 := ObjectA1;
1193          HeadA2 := ObjectA2;
1194          HeadB1 := ObjectB1;
- 1195          HeadB2 := ObjectB2;
1196          HeadC1 := ObjectC1;
1197          HeadC2 := ObjectC2;
1198
- 1199          while Current^.Next^.Next <> nil          { Skip G1, G2.}
1200              do begin

```

```

1201      new(ObjectA1^.Next);
1202      ObjectA1 := ObjectA1^.Next;
1203      ObjectA1^.Node := Current^.Node;
1204      ObjectA1^.Next := nil;
1205      Current := Current^.Next;
1206
1207      new(ObjectA2^.Next);
1208      ObjectA2 := ObjectA2^.Next;
1209      ObjectA2^.Node := Current^.Node;
1210      ObjectA2^.Next := nil;
1211      Current := Current^.Next;
1212
1213      new(ObjectB1^.Next);
1214      ObjectB1 := ObjectB1^.Next;
1215      ObjectB1^.Node := Current^.Node;
1216      ObjectB1^.Next := nil;
1217      Current := Current^.Next;
1218
1219      new(ObjectB2^.Next);
1220      ObjectB2 := ObjectB2^.Next;
1221      ObjectB2^.Node := Current^.Node;
1222      ObjectB2^.Next := nil;
1223      Current := Current^.Next;
1224
1225      new(ObjectC1^.Next);
1226      ObjectC1 := ObjectC1^.Next;
1227      ObjectC1^.Node := Current^.Node;
1228      ObjectC1^.Next := nil;
1229      Current := Current^.Next;
1230      new(ObjectC2^.Next);
1231      ObjectC2 := ObjectC2^.Next;
1232      ObjectC2^.Node := Current^.Node;
1233      ObjectC2^.Next := nil;
1234      Current := Current^.Next;
1235
1236      end;
1237
1238      ObjectA1 := HeadA1;
1239      ObjectA2 := HeadA2;
1240      ObjectB1 := HeadB1;
1241      ObjectB2 := HeadB2;
1242      ObjectC1 := HeadC1;
1243      ObjectC2 := HeadC2;
1244
1245      if (i = i div 2 * 2)
1246      then if GrownDeadNode(ObjectA2, Data) or
1247           GrownDeadNode(ObjectB2, Data) or
1248           GrownDeadNode(ObjectC2, Data)
1249           then DeadNode := true
1250           else DeadNode := false

```



```

1251         else if GrownDeadNode(ObjectA1, Data) or
1252             GrownDeadNode(ObjectB1, Data) or
1253             GrownDeadNode(ObjectC1, Data)
1254         then DeadNode := true
1255             else DeadNode := false;
1256     end;
1257
1258
1259
1260
1261
1262
1263
1264 begin                                     ( BuildGraph )
1265     rewrite(VERTICES);
1266
1267     n := 28;    { Dimension of Cost[n,n] }
1268     for i := 1 to n do
1269     for j := 1 to n do
1270         A[i,j] := 9999.99;    ( 9999.99 means the infinity )
1271
1272     From := List;
1273     To := List;
1274     i := 0;
1275     j := 0;
1276
1277     while From <> nil do
1278     begin
1279         i := i + 1;
1280         if DeadNode(List, From^.Node, i)
1281         then { nothing }
1282         else begin
1283             To := List;
1284             j := 0;
1285             while To <> nil do
1286             begin
1287                 j := j + 1;
1288                 if (From = To) or
1289                     CrossVertices(List, From^.Node, To^.Node)
1290                     DeadNode(List, From^.Node, j) or
1291                     DeadNode(List, To^.Node, j) or
1292                     Interference(List, From^.Node, To^.Node,
1293                     then { nothing }
1294                     else A[i,j] := sqrt(
1295                                     (From^.Node.x - To^.Node.x)
1296                                     (From^.Node.x - To^.Node.x)
1297                                     (From^.Node.y - To^.Node.y)
1298                                     (From^.Node.y - To^.Node.y)
1299
1300             To := To^.Next;
1301         end;

```

```

1301         end;
1302         From := From^.Next;
1303     end;
1304
1305     PrintAMatrix(A);
1306
1307 end; { Of Procedure BuildVGraph }
1308
1309
1310
1311
1312
1313
1314
1315
1316 procedure SearchVGraph(A : CostMatrix; var LinkedPath : PathType);
1317 (
1318
1319     Author   :   C. H. Chung
1320
1321     Version  :   2.7
1322
1323     Date     :   November 7, 1988
1324
1325 )
1326
1327     This program will calculate the shortest path by Floyd Algorithm.
1328
1329     SearchVGraph(A, LinkedPath)
1330         . To find the shortest path of the VGraph
1331         . LinkedPath holds the information of the shortest path
1332         . by the VGraph Algorithm.
1333         . INPUT FILE : _____
1334         . OUTPUT FILE : PATH
1335         . input of this procedure : A
1336         . output of this procedure : LinkedPath
1337
1338
1339
1340     LinkedPath : PathType; *** output of procedure SearchVGraph ***
1341     /* Linked LinkedPath for the shortest path */
1342     A : CostMatrix; /* A : Cost matrix for Floyd Algorithm */
1343     P : PathMatrix; /* P : Path Matrix for Floyd Algorithm */
1344     n : integer; /* Dimension of Cost (Path) Matrix */
1345     Cost : real; /* Cost of the shortest path */
1346
1347
1348
1349 type
1350     PathMatrix = array [1..28,1..28] of integer;

```

```

1351 var
1352     P : PathMatrix;      { Path Matrix for the Floyd Algorithm }
1353     n : integer;         { Dimension of the Cost (Path) Matrix }
1354     Cost : real;         { Cost of the shortest path }
1355
1356
1357
1358
1359
1360
1361
1362     procedure InitializePath(var P : PathMatrix;
1363                             var LinkedPath : PathType;
1364                             var n : integer);
1365     (
1366         This procedure will initialize the Cost Matrix and Path Matrix.
1367         The Path Matrix are automatically set to zero.
1368         The Cost Matrix should be defined by User.
1369         The Start node and Goal node should be defined by User
1370
1371         9999 means the infinitive.
1372
1373     var
1374         i, j : integer;
1375         Start, Goal : integer;
1376
1377     begin
1378         n := 28;
1379         Start := 1;
1380         Goal := 27;
1381         (*****
1382           { Start, Goal }
1383           *****)
1384         for i := 1 to n do
1385             for j := 1 to n do
1386                 P[i,j] := 0;
1387
1388         LinkedPath := nil;
1389         new(LinkedPath);
1390         LinkedPath^.Data := Start;
1391         LinkedPath^.Next := nil;
1392         new(LinkedPath^.Next);
1393         LinkedPath^.Next^.Data := Goal;
1394         LinkedPath^.Next^.Next := nil
1395     end;
1396
1397
1398
1399
1400

```

```

1401 procedure PrintPath(LinkedPath : PathType);
1402 {
1403   This procedure will print the Linked LinkedPath of the shortest
1404   path.
1405 }
1406 var
1407   Current : PathType;
1408 begin
1409   write(PATH, ' Path represented by internal nodes = ');
1410   Current := LinkedPath;
1411   write(PATH, Current^.Data :7);
1412   Current := Current^.Next;
1413   while (Current<> nil)
1414     do begin
1415       write(PATH, Current^.Data :7);
1416       Current := Current^.Next;
1417     end;
1418   writeln(PATH);
1419 end;
1420
1421
1422
1423
1424
1425
1426
1427 procedure WriteVertice(i : integer);
1428 {
1429   This procedure prints vertice.
1430 }
1431 begin
1432   case i of
1433     1 : write(PATH, ' START in 0` sliced');
1434     2 : write(PATH, ' START in 90` sliced');
1435     3 : write(PATH, ' A1 in 0` sliced');
1436     9 : write(PATH, ' A2 in 0` sliced');
1437    15 : write(PATH, ' A3 in 0` sliced');
1438    21 : write(PATH, ' A4 in 0` sliced');
1439     5 : write(PATH, ' B1 in 0` sliced');
1440    11 : write(PATH, ' B2 in 0` sliced');
1441    17 : write(PATH, ' B3 in 0` sliced');
1442    23 : write(PATH, ' B4 in 0` sliced');
1443     7 : write(PATH, ' C1 in 0` sliced');
1444    13 : write(PATH, ' C2 in 0` sliced');
1445    19 : write(PATH, ' C3 in 0` sliced');
1446    25 : write(PATH, ' C4 in 0` sliced');
1447     4 : write(PATH, ' A1 in 90` sliced');
1448    10 : write(PATH, ' A2 in 90` sliced');
1449    16 : write(PATH, ' A3 in 90` sliced');
1450    22 : write(PATH, ' A4 in 90` sliced');

```



```

1501 WriteVertice(j);
1502 write(PATH, ' ', A[i,j] :7:3);
1503 case Flag of
1504   -1 : writeln(PATH, ' <-90`>');
1505   -0 : writeln(PATH, ' <0`>');
1506   1 : writeln(PATH, ' <90`>');
1507 end;
1508 Current := Current^.Next;
1509
1510 while (Current^.Next <> nil)
1511   do begin
1512     i := Current^.Data;
1513     j := Current^.Next^.Data;
1514     if (Current^.Next^.Next = nil) or
1515       (Current^.Next^.Data > 26)
1516     then j := Current^.Next^.Data;
1517     WriteVertice(i);
1518     write(PATH, ' -----> ');
1519     WriteVertice(j);
1520     write(PATH, ' ', A[i,Current^.Next^.Data] :7:3);
1521     first := Current^.Data mod 2;
1522     second := Current^.Next^.Data mod 2;
1523     Flag := first - second;
1524     case Flag of
1525       -1 : writeln(PATH, ' <-90`>');
1526       0 : writeln(PATH, ' <0`>');
1527       1 : writeln(PATH, ' <90`>');
1528     end;
1529     Current := Current^.Next;
1530   end;
1531 end;
1532
1533
1534
1535
1536
1537
1538
1539 procedure PrintCostAndPath(A : CostMatrix; LinkedPath : PathType;
1540                             Cost : real);
1541 (
1542   This procedure will print the shortest path and its cost.
1543 )
1544 var
1545   Current : PathType;
1546   Start, Goal : integer;
1547
1548 begin
1549   Current := LinkedPath;
1550   Start := Current^.Data;

```

```

1551     while (Current^.Next <> nil)
1552     do Current := Current^.Next;
1553     Goal := Current^.Data;
1554
1555     writeln(PATH);
1556     writeln(PATH);
1557     writeln(PATH);
1558     writeln(PATH);
1559     write(PATH, '_____');
1560     writeln(PATH, '_____');
1561     writeln(PATH);
1562     write(PATH, ' The shortest path is calculated by');
1563     writeln(PATH, ' the Graph Search Algorithm. ');
1564     writeln(PATH);
1565     writeln(PATH, '                Start Node = ', Start :3,
1566                '                Goal Node = ', Goal :3);
1567     writeln(PATH);
1568     write(PATH, '_____');
1569     writeln(PATH, '_____');
1570     writeln(PATH);
1571     PrintPrettyLinkedPath(A, LinkedPath);
1572     writeln(PATH);
1573     write(PATH, '_____');
1574     writeln(PATH, '_____');
1575     writeln(PATH);
1576     writeln(PATH, '                The Total Cost = ', Cost :6:3);
1577     write(PATH, '_____');
1578     writeln(PATH, '_____');
1579     writeln(PATH);
1580     writeln(PATH);
1581 end;
1582
1583
1584
1585
1586
1587
1588
1589 procedure FindPath(var P : PathMatrix; var LinkedPath : PathType
1590 {
1591     This procedure will find the shortest path from Path Matrix.
1592 }
1593 var
1594     i, j : integer;
1595     Current : PathType;
1596
1597 begin
1598     i := LinkedPath^.Data;
1599     j := LinkedPath^.Next^.Data;
1600     Current := nil;

```

```

1601         if P[i,j] = 0
1602             then
1603                 else begin
1604                     new(Current);
1605                     Current^.Data := P[i,j];
1606                     Current^.Next := LinkedPath^.Next;
1607                     LinkedPath^.Next := Current;
1608                     FindPath(P, LinkedPath);
1609                     FindPath(P, LinkedPath^.Next);
1610                 end;
1611     end;
1612
1613
1614
1615
1616
1617
1618
1619     procedure FindCost(A : CostMatrix; LinkedPath : PathType;
1620                       var Cost : real);
1621     (
1622         This procedure will find the cost of the shortest path
1623         from the Cost Matrix.
1624     )
1625     var
1626         Current : PathType;
1627         i, j : integer;
1628
1629     begin
1630         Current := LinkedPath;
1631         Cost := 0;
1632
1633         while (Current^.Next <> nil)
1634             do begin
1635                 i := Current^.Data;
1636                 j := Current^.Next^.Data;
1637                 Cost := Cost + A[i,j];
1638                 Current := Current^.Next;
1639             end;
1640     end;
1641
1642
1643
1644
1645
1646
1647
1648     procedure CalculateCostAndPath(var A : CostMatrix;
1649                                   var P : PathMatrix;
1650                                   n : integer);

```


This procedure will calculate the Cost Matrix and the Path Matrix by the Floyd Algorithm.

A : Cost Matrix
P : Path Matrix

```
var
  order : integer;
  i, j : integer;
  value : real;

begin
  order := 1;
  repeat
    for i := 1 to n do
      for j := 1 to n do
        if ((i <> order) and (j <> order))
          then begin
            value := A[i,order] + A[order,j];
            if (A[i,j] > value)
              then begin
                A[i,j] := value;
                P[i,j] := order;
              end;
          end;
        end;
      order := order + 1;
    until not (order <= n)
  end;
```

```
begin (____ Procedure SearchVGraph ____)
```

```
  rewrite(PATH);
```

```
  InitializePath(P, LinkedPath, n);
```

```
  CalculateCostAndPath(A, P, n);
```

```
  FindPath(P, LinkedPath);
```

```
  FindCost(A, LinkedPath, Cost);
```

```
  PrintCostAndPath(A, LinkedPath, Cost);
```

```
end; (____ Of procedure SearchVGraph ____)
```

```
1701
1702
1703
1704
1705
1706 begin (_____ Main _____)
1707
1708     BuildGSpaceObstacles(List);
1709     BuildVGraph(A, List);
1710     SearchVGraph(A, LinkedPath);
1711
1712 end. (_____ of Main _____)
1713
1714
```

Appendix B: I/O FIELDS for the VGraph Algorithm

[OBSTACLES]

3.0000	18.0000
9.0000	18.0000
9.0000	10.0000
3.0000	10.0000
10.5000	13.0000
19.0000	13.0000
19.0000	9.0000
10.5000	9.0000
8.0000	7.5000
16.0000	7.5000
16.0000	3.0000
8.0000	3.0000

[GSPACE]

1.0000	18.0000
2.0000	18.0000
8.5000	13.0000
9.5000	13.0000
6.0000	7.5000
7.0000	7.5000

9.0000	18.0000
9.0000	18.0000
19.0000	13.0000
19.0000	13.0000
16.0000	7.5000
16.0000	7.5000

9.0000	9.0000
9.0000	8.0000
19.0000	8.0000
19.0000	7.0000
16.0000	2.0000
16.0000	1.0000

1.0000	9.0000
2.0000	8.0000
8.5000	8.0000
9.5000	7.0000
6.0000	2.0000
7.0000	1.0000

[VERTICES]

A[1, 2] = 0.0000
 A[1, 7] = 1.1180
 A[1, 8] = 2.0616
 A[1,16] = 4.1231
 A[1,21] = 4.4721
 A[1,22] = 3.1623
 A[1,25] = 5.0990
 A[1,26] = 6.3246

A[2, 1] = 0.0000
 A[2, 7] = 1.1180
 A[2, 8] = 2.0616
 A[2,16] = 4.1231
 A[2,21] = 4.4721
 A[2,22] = 3.1623
 A[2,25] = 5.0990
 A[2,26] = 6.3246

A[3, 4] = 1.0000
 A[3, 9] = 8.0000
 A[3,10] = 8.0000
 A[3,21] = 9.0000
 A[3,22] = 10.0499

A[4, 3] = 1.0000
 A[4, 9] = 7.0000
 A[4,10] = 7.0000
 A[4,21] = 9.0554
 A[4,22] = 10.0000

A[6, 9] = 5.0249
 A[6,10] = 5.0249
 A[6,11] = 9.5000
 A[6,12] = 9.5000
 A[6,16] = 5.0249
 A[6,17] = 10.7355
 A[6,27] = 4.0311
 A[6,28] = 4.0311

A[7, 1] = 1.1180
 A[7, 2] = 1.1180
 A[7, 8] = 1.0000
 A[7,13] = 10.0000
 A[7,16] = 3.0414
 A[7,17] = 13.0096
 A[7,21] = 5.2202
 A[7,22] = 4.0311
 A[7,23] = 2.5495

A[7,25] = 5.5000
A[7,26] = 6.5765

A[8, 2] = 2.0616
A[8, 7] = 1.0000
A[8,13] = 9.0000
A[8,16] = 2.0616
A[8,17] = 12.0104
A[8,19] = 10.5475
A[8,21] = 6.1847
A[8,22] = 5.0249
A[8,23] = 1.5811
A[8,25] = 5.5902
A[8,26] = 6.5000

A[9, 3] = 8.0000
A[9, 4] = 7.0000
A[9, 6] = 5.0249
A[9,10] = 0.0000
A[9,11] = 11.1803
A[9,12] = 11.1803
A[9,16] = 10.0000
A[9,27] = 5.0000
A[9,28] = 5.0000

A[10, 3] = 8.0000
A[10, 4] = 7.0000
A[10, 6] = 5.0249
A[10, 9] = 0.0000
A[10,11] = 11.1803
A[10,12] = 11.1803
A[10,16] = 10.0000
A[10,27] = 5.0000
A[10,28] = 5.0000

A[11, 6] = 9.5000
A[11, 9] = 11.1803
A[11,10] = 11.1803
A[11,12] = 0.0000
A[11,17] = 5.0000
A[11,18] = 6.0000
A[11,27] = 6.3246
A[11,28] = 6.3246

A[12, 6] = 9.5000
A[12, 9] = 11.1803
A[12,10] = 11.1803
A[12,11] = 0.0000
A[12,17] = 5.0000
A[12,18] = 6.0000
A[12,27] = 6.3246
A[12,28] = 6.3246

A[13, 7] = 10.0000
 A[13, 17] = 3.0414
 A[13, 19] = 5.5000
 A[13, 23] = 7.5166

A[16, 2] = 4.1231
 A[16, 6] = 5.0249
 A[16, 7] = 3.0414
 A[16, 8] = 2.0616
 A[16, 10] = 10.0000
 A[16, 11] = 11.1803
 A[16, 13] = 7.0178
 A[16, 17] = 10.0000
 A[16, 22] = 7.0000
 A[16, 23] = 0.5000

A[17, 6] = 10.7355
 A[17, 7] = 13.0096
 A[17, 11] = 5.0000
 A[17, 12] = 5.0000
 A[17, 13] = 3.0414
 A[17, 18] = 1.0000
 A[17, 19] = 6.7082
 A[17, 23] = 10.5000

A[18, 11] = 6.0000
 A[18, 12] = 6.0000
 A[18, 13] = 3.0414
 A[18, 17] = 1.0000
 A[18, 19] = 5.8310
 A[18, 20] = 6.7082

A[19, 13] = 5.5000
 A[19, 17] = 6.7082
 A[19, 18] = 5.8310
 A[19, 20] = 1.0000
 A[19, 25] = 10.0000

A[20, 13] = 6.5000
 A[20, 17] = 7.6158
 A[20, 18] = 6.7082
 A[20, 19] = 1.0000
 A[20, 25] = 10.0499
 A[20, 26] = 9.0000

A[21, 1] = 4.4721
 A[21, 3] = 9.0000
 A[21, 4] = 9.0554
 A[21, 7] = 5.2202
 A[21, 22] = 1.4142
 A[21, 23] = 7.5664
 A[21, 25] = 8.6023
 A[21, 26] = 10.0000

A[22, 1] = 3.1623
 A[22, 2] = 3.1623
 A[22, 4] = 10.0000
 A[22, 7] = 4.0311
 A[22, 8] = 5.0249
 A[22, 13] = 14.0089
 A[22, 16] = 7.0000
 A[22, 17] = 17.0000
 A[22, 21] = 1.4142
 A[22, 23] = 6.5000
 A[22, 25] = 7.2111
 A[22, 26] = 8.6023

A[23, 2] = 3.6401
 A[23, 7] = 2.5495
 A[23, 8] = 1.5811
 A[23, 10] = 10.0125
 A[23, 13] = 7.5166
 A[23, 16] = 0.5000
 A[23, 17] = 10.5000
 A[23, 21] = 7.5664
 A[23, 22] = 6.5000

A[25, 1] = 5.0990
 A[25, 2] = 5.0990
 A[25, 7] = 5.5000
 A[25, 8] = 5.5902
 A[25, 19] = 10.0000
 A[25, 21] = 8.6023
 A[25, 22] = 7.2111
 A[25, 26] = 1.4142

A[26, 2] = 6.3246
 A[26, 8] = 6.5000
 A[26, 19] = 9.0554
 A[26, 20] = 9.0000
 A[26, 22] = 8.6023
 A[26, 25] = 1.4142

A[27, 6] = 4.0311
 A[27, 9] = 5.0000
 A[27, 10] = 5.0000
 A[27, 11] = 6.3246
 A[27, 12] = 6.3246
 A[27, 28] = 0.0000

A[28, 6] = 4.0311
 A[28, 9] = 5.0000
 A[28, 10] = 5.0000
 A[28, 11] = 6.3246
 A[28, 12] = 6.3246
 A[28, 27] = 0.0000

[PATH]

The shortest path is calculated by the Graph Search Algorithm.

Start Node = 1 Goal Node = 27

Path represented by internal nodes = 1 16 6 27

From	To	Cost	Rotation
START in 0' sliced ----->	A3 in 90' sliced	4.123	<90'>
A3 in 90' sliced ----->	B1 in 90' sliced	5.025	<0'>
B1 in 90' sliced ----->	GOAL in 0' sliced	4.031	<-90'>

The Total Cost = 13.179

Appendix C: Simulation of the Rotational GSpace

```
1 program BuildGrownSpaceObstaclesWithRotation(OBSTACLES, ROTATION); --
2 {
3
4     Author   :   C. H. Chung
5
6     Version  :   3.5
7
8     Date     :   November 17, 1988
9
10
11
12     BuildGSpaceWithRotation;
13
14         .   To build the Grown Space Obstacles with rotation
15         .   INPUT  FILE : OBSTACLES
16         .   OUTPUT FILE : ROTATION
17
18
19     This program will build the Grown Space Obstacles.
20
21 type
22     Point2D = record
23         x, y : real;
24     end;
25     Vertice2D = ^Node2D;
26     Node2D = record
27         Node : Point2D;
28         Next : Vertice2D
29     end;
30 var
31     OBSTACLES, ROTATION : text;
32
33
34
35
36
37     procedure BuildGSpaceWithRotation;
38     (
39
40         Author   :   C. H. Chung
41
42         Version  :   2.3
43
44         Date     :   December 3, 1988
45
46
47
48         BuildGSpaceWithRotation;
49
50         .   To build the Grown Space Obstacles with rotation
```

```

51      . INPUT  FILE : OBSTACLES
52      . OUTPUT FILE : ROTATION
53
54
55      This program will build the Grown Space Obstacles.
56
57      hh : the horizontal length of the object
58
59      vv : the vertical length of the object
60
61      rr : the sliced angle for rotational Grown Space Obstacles.
62
63  const
64      Pi = 3.141592;      (Radian)
65  var
66      Object : Vertice2D;
67      ObjectA, ObjectB, ObjectC : Vertice2D;
68      hh, vv, rr : real;
69
70
71
72
73
74  procedure Print2Dvertice(List : Vertice2D);
75  (
76      This procedure will print the Linked List of the
77      shortest path.
78  )
79  begin
80      if List = nil
81      then writeln(ROTATION)
82      else begin
83          writeln(ROTATION, List^.Node.x :10:4,
84                  List^.Node.y :10:4);
85          Print2Dvertice(List^.Next)
86      end
87  end;
88
89
90
91
92
93
94  procedure CreateObject(var Object : Vertice2D);
95  (
96      This procedure creates the object from the input file
97      by the linked list.
98  )
99  var
100     Current : Vertice2D;

```

```

101 begin
102     Object := nil;
103     if not eof(OBSTACLES)
104         then begin
105             new(Object);
106             readln(OBSTACLES, Object^.Node.x,
107                 Object^.Node.y);
108             Object^.Next := nil;
109             Current := Object;
110             while not eof(OBSTACLES)
111                 do begin
112                     new(Current^.Next);
113                     Current := Current^.Next;
114                     readln(OBSTACLES, Current^.Node.x,
115                         Current^.Node.y);
116                     Current^.Next := nil
117                 end
118             end
119         end;
120
121
122
123
124
125
126
127 procedure GrownObject(var Object, Grown : Vertice2D;
128                       hh, vv, rr : real);
129 (
130     This procedure builds the Grown Space Obstacles.
131
132     where h : horizontal length
133           v : vertical length
134
135     0 < q < Pi/2
136
137         a1 = (A1x,A1y) + h(-cos(q),-sin(q))
138         a2 = (A1x,A1y)
139         a3 = (A2x,A2y)
140         a4 = (A2x,A2y) + v(sin(q),-cos(q))
141         a5 = (A3x,A3y) + v(sin(q),-cos(q))
142         a6 = (a5x,a5y) + h(-cos(q),-sin(q))
143         a8 = (A4x,A4y) + h(-cos(q),-sin(q))
144         a7 = (a8x,a8y) + v(sin(q),-cos(q))
145
146     Pi/2 < q < Pi
147
148         q = q - Pi/2
149         temp = h      (to swap h and v)
150         h = v

```

```

151         v = temp
152
153         q = 0
154         Delete a2, a4, a6, a8.
155
156         q = Pi/2
157         Swap h and v.
158         Delete a2, a4, a6, a8.
159
160     var
161         Current, Head : Vertice2D;
162     begin
163         Current := nil;
164         new(Current);
165         Current^.Node.x := Object^.Node.x - hh * cos(rr);
166         Current^.Node.y := Object^.Node.y - hh * sin(rr);
167         Current^.Next := nil;
168
169         Head := Object;
170         Grown := Current;
171
172         new(Current^.Next);
173         Current := Current^.Next;
174         Current^.Node.x := Object^.Node.x;
175         Current^.Node.y := Object^.Node.y;
176         Current^.Next := nil;
177
178         Object := Object^.Next;
179         new(Current^.Next);
180         Current := Current^.Next;
181         Current^.Node.x := Object^.Node.x;
182         Current^.Node.y := Object^.Node.y;
183         Current^.Next := nil;
184
185         new(Current^.Next);
186         Current := Current^.Next;
187         Current^.Node.x := Object^.Node.x + vv * sin(rr);
188         Current^.Node.y := Object^.Node.y - vv * cos(rr);
189         Current^.Next := nil;
190
191         Object := Object^.Next;
192         new(Current^.Next);
193         Current := Current^.Next;
194         Current^.Node.x := Object^.Node.x + vv * sin(rr);
195         Current^.Node.y := Object^.Node.y - vv * cos(rr);
196         Current^.Next := nil;
197
198         new(Current^.Next);
199         Current := Current^.Next;
200         Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr);

```

```

201 Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr);
202 Current^.Next := nil;
203
204 Object := Object^.Next;
205 new(Current^.Next);
206 Current := Current^.Next;
207 Current^.Node.x := Object^.Node.x + vv * sin(rr) - hh * cos(rr);
208 Current^.Node.y := Object^.Node.y - vv * cos(rr) - hh * sin(rr);
209 Current^.Next := nil;
210
211 new(Current^.Next);
212 Current := Current^.Next;
213 Current^.Node.x := Object^.Node.x - hh * cos(rr);
214 Current^.Node.y := Object^.Node.y - hh * sin(rr);
215 Current^.Next := nil;
216
217 Object := Head;
218
219 if (rr = 0) or (abs(rr - Pi/2) < 0.001)
220 then begin
221     Current := nil;
222     new(Current);
223     Current^.Node := Grown^.Node;
224     Current^.Next := nil;
225
226     Head := nil;
227     Head := Current;
228     while Grown^.Next^.Next <> nil
229     do begin
230         new(Current^.Next);
231         Current := Current^.Next;
232         Grown := Grown^.Next^.Next;
233         Current^.Node := Grown^.Node;
234         Current^.Next := nil;
235     end;
236     Grown := Head;
237 end;
238 end;
239
240
241
242
243
244
245 procedure RotationObjects(ObjectA, ObjectB, ObjectC : Vertice2D;
246 rr, hh, vv : real);
247
248 {
249     This procedure will print the GSpace Obstacle with Rotation.
250 }
251 var

```

```

251      GrownA, GrownB, GrownC : Vertice2D;
252      i : integer;
253      Angle : real;
254  begin
255      i := 0;
256      Angle := rr * i;
257
258      while (Angle >= 0) and (Angle < Pi) do
259          begin
260              writeln(ROTATION, Angle*180/Pi :5:1, ' Rotation');
261              if Angle = 0
262                  then begin
263                      GrownObject(ObjectA, GrownA, hh, vv, Angle);
264                      GrownObject(ObjectB, GrownB, hh, vv, Angle);
265                      GrownObject(ObjectC, GrownC, hh, vv, Angle);
266                      Print2Dvertice(GrownA);
267                      Print2Dvertice(GrownB);
268                      Print2Dvertice(GrownC);
269                  end;
270              if (Angle > 0) and (Angle < Pi/2)
271                  then begin
272                      GrownObject(ObjectA, GrownA, hh, vv, Angle);
273                      GrownObject(ObjectB, GrownB, hh, vv, Angle);
274                      GrownObject(ObjectC, GrownC, hh, vv, Angle);
275                      Print2Dvertice(GrownA);
276                      Print2Dvertice(GrownB);
277                      Print2Dvertice(GrownC);
278                  end;
279              if abs(Angle - Pi/2) < 0.001 ( because of Round Off )
280                  then begin
281                      GrownObject(ObjectA, GrownA, vv, hh, Angle-Pi/2);
282                      GrownObject(ObjectB, GrownB, vv, hh, Angle-Pi/2);
283                      GrownObject(ObjectC, GrownC, vv, hh, Angle-Pi/2);
284                      Print2Dvertice(GrownA);
285                      Print2Dvertice(GrownB);
286                      Print2Dvertice(GrownC);
287                  end
288              else if (Angle > Pi/2)
289                  then begin
290                      GrownObject(ObjectA, GrownA, vv, hh, Angle-Pi/2);
291                      GrownObject(ObjectB, GrownB, vv, hh, Angle-Pi/2);
292                      GrownObject(ObjectC, GrownC, vv, hh, Angle-Pi/2);
293                      Print2Dvertice(GrownA);
294                      Print2Dvertice(GrownB);
295                      Print2Dvertice(GrownC);
296                  end;
297              i := i + 1;
298              Angle := rr * i;
299          end;
300      end;

```

```

301
302
303
304
305
306 procedure Partition(var Object, ObjectA, ObjectB,
307                      ObjectC : Vertice2D);
308 (
309     This procedure will partition the whole Object into
310     3 objects.
311 )
312 var
313     i : integer;
314     Current, CurrentA, CurrentB, CurrentC : Vertice2D;
315 begin
316     Current := Object;
317
318     ObjectA := nil;
319     new(ObjectA);
320     ObjectA^.Node := Current^.Node;
321     ObjectA^.Next := nil;
322     CurrentA := ObjectA;
323     for i := 1 to 3 do
324         begin
325             new(CurrentA^.Next);
326             Current := Current^.Next;
327             CurrentA := CurrentA^.Next;
328             CurrentA^.Node := Current^.Node;
329             CurrentA^.Next := nil;
330         end;
331
332     Current := Current^.Next;
333
334     ObjectB := nil;
335     new(ObjectB);
336     ObjectB^.Node := Current^.Node;
337     ObjectB^.Next := nil;
338     CurrentB := ObjectB;
339     for i := 1 to 3 do
340         begin
341             new(CurrentB^.Next);
342             Current := Current^.Next;
343             CurrentB := CurrentB^.Next;
344             CurrentB^.Node := Current^.Node;
345             CurrentB^.Next := nil;
346         end;
347
348     Current := Current^.Next;
349
350     ObjectC := nil;

```



```

351     new(ObjectC);
352     ObjectC^.Node := Current^.Node;
353     ObjectC^.Next := nil;
354     CurrentC := ObjectC;
355     for i := 1 to 3 do
356         begin
357             new(CurrentC^.Next);
358             Current := Current^.Next;
359             CurrentC := CurrentC^.Next;
360             CurrentC^.Node := Current^.Node;
361             CurrentC^.Next := nil;
362         end;
363     end; { of procedure Partition }
364
365
366
367
368     begin {_____ BuildGSpaceWithRotation _____}
369         reset(OBSTACLES);
370         rewrite(ROTATION);
371
372         rr := Pi / 6;
373         hh := 2;
374         vv := 1;
375
376         CreateObject(Object);
377         Partition(Object, ObjectA, ObjectB, ObjectC);
378         RotationObjects(ObjectA, ObjectB, ObjectC, rr, hh, vv);
379
380     end; { of Procedure BuildGSpaceWithRotation }
381
382
383
384
385
386 begin {_____ Main _____}
387
388     BuildGSpaceWithRotation;
389
390 end. {___ Of Main ___}

```

Appendix D: I/O FILES for the Rotational GSpace

[OBSTACLES]

3.0000	18.0000
9.0000	18.0000
9.0000	10.0000
3.0000	10.0000
10.5000	13.0000
19.0000	13.0000
19.0000	9.0000
10.5000	9.0000
8.0000	7.5000
16.0000	7.5000
16.0000	3.0000
8.0000	3.0000

[ROTATION]

0.0° Rotation

1.0000	18.0000
9.0000	18.0000
9.0000	9.0000
1.0000	9.0000
8.5000	13.0000
19.0000	13.0000
19.0000	8.0000
8.5000	8.0000
6.0000	7.5000
16.0000	7.5000
16.0000	2.0000
6.0000	2.0000

30.0° Rotation

1.2679	17.0000
3.0000	18.0000
9.0000	18.0000
9.5000	17.1340
9.5000	9.1340
7.7679	8.1340
1.7679	8.1340
1.2679	9.0000

8.7679	12.0000
10.5000	13.0000
19.0000	13.0000
19.5000	12.1340
19.5000	8.1340
17.7679	7.1340
9.2679	7.1340
8.7679	8.0000

6.2679	6.5000
8.0000	7.5000
16.0000	7.5000
16.5000	6.6340
16.5000	2.1340
14.7679	1.1340
6.7679	1.1340
6.2679	2.0000

60.0` Rotation

2.0000	16.2679
3.0000	18.0000
9.0000	18.0000
9.8660	17.5000
9.8660	9.5000
8.8660	7.7679
2.8660	7.7679
2.0000	8.2679

9.5000	11.2679
10.5000	13.0000
19.0000	13.0000
19.8660	12.5000
19.8660	8.5000
18.8660	6.7679
10.3660	6.7679
9.5000	7.2679

7.0000	5.7679
8.0000	7.5000
16.0000	7.5000
16.8660	7.0000
16.8660	2.5000
15.8660	0.7679
7.8660	0.7679
7.0000	1.2679

90.0' Rotation

2.0000	18.0000
3.0000	18.0000
9.0000	18.0000
9.0000	16.0000
9.0000	8.0000
8.0000	8.0000
2.0000	8.0000
2.0000	10.0000

9.5000	13.0000
10.5000	13.0000
19.0000	13.0000
19.0000	11.0000
19.0000	7.0000
18.0000	7.0000
9.5000	7.0000
9.5000	9.0000

7.0000	7.5000
8.0000	7.5000
16.0000	7.5000
16.0000	5.5000
16.0000	1.0000
15.0000	1.0000
7.0000	1.0000
7.0000	3.0000

120.0' Rotation

2.1340	17.5000
3.0000	18.0000
9.0000	18.0000
10.0000	16.2679
10.0000	8.2679
9.1340	7.7679
3.1340	7.7679
2.1340	9.5000

9.6340	12.5000
10.5000	13.0000
19.0000	13.0000
20.0000	11.2679
20.0000	7.2679
19.1340	6.7679
10.6340	6.7679
9.6340	8.5000

7.1340	7.0000
8.0000	7.5000
16.0000	7.5000
17.0000	5.7679
17.0000	1.2679
16.1340	0.7679
8.1340	0.7679
7.1340	2.5000

150.0` Rotation

2.5000	17.1340
3.0000	18.0000
9.0000	18.0000
10.7321	17.0000
10.7321	9.0000
10.2321	8.1340
4.2321	8.1340
2.5000	9.1340

10.0000	12.1340
10.5000	13.0000
19.0000	13.0000
20.7321	12.0000
20.7321	8.0000
20.2321	7.1340
11.7321	7.1340
10.0000	8.1340

7.5000	6.6340
8.0000	7.5000
16.0000	7.5000
17.7321	6.5000
17.7321	2.0000
17.2321	1.1340
9.2321	1.1340
7.5000	2.1340

Appendix E: Simulation of the Branch and Bound Algorithm

```

1  program BranchAndBoundAlgorithm(BBinput, BBoutput);
2  (
3
4      Author   :   C. H. Chung
5
6      Version  :   2.0
7
8      ate      :   November 1, 1988
9
10
11  NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
12  This NodeSet is implemented by linked list, which node has
13              the record structure to represent the vertices.
14  Input file comes from BBinput.
15  Output file is BBoutput.
16  )
17
18  type
19
20      PointType = record
21          x, y, z : real
22      end;
23
24      NodeType = ^Nodes;
25      Nodes = record
26          Node : PointType;
27          Next : NodeType
28      end;
29
30  var
31
32      BBinput, BBoutput : text;
33      NodeSet, MinSet : NodeType;
34
35
36
37
38
39
40
41  procedure PrintNodes(NodeSet : NodeType);
42  (
43      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
44      This Procedure will print the NodeSet.
45  )
46  begin
47      if NodeSet = nil
48      then
49      else begin
50          writeln(BBoutput, NodeSet^.Node.x :10:4,

```

```

51                                     NodeSet^.Node.y :10:4,
52                                     NodeSet^.Node.z :10:4);
53                               writeln(BBoutput);
54                               PrintNodes(NodeSet^.Next)
55                               end
56   end;
57
58
59
60
61
62
63   function EuclideanDistance(NodeSet : NodeType) : real;
64   {
65       NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
66       This Function will calculate the Euclidean Distance
67       between the points in 3D.
68   }
69   var
70       Current : NodeType;
71       d1, d2, d3 : real;
72       x1, y1, z1,
73       x2, y2, z2 : real;
74   begin
75       Current := NodeSet;
76       x1 := Current^.Node.x;
77       y1 := Current^.Node.y;
78       z1 := Current^.Node.z;
79
80       Current := Current^.Next;
81       x2 := Current^.Node.x;
82       y2 := Current^.Node.y;
83       z2 := Current^.Node.z;
84
85       d1 := (x2 - x1) * (x2 - x1);
86       d2 := (y2 - y1) * (y2 - y1);
87       d3 := (z2 - z1) * (z2 - z1);
88
89       EuclideanDistance := sqrt(d1 + d2 + d3)
90   end;
91
92
93
94
95
96
97   function LengthOfNodeSet(NodeSet : NodeType) : integer;
98   {
99       NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph...
100      This Function will find the length of NodeSet.

```

```

101
102 begin
103     if (NodeSet^.Next = nil)
104         then LengthOfNodeSet := 1
105         else LengthOfNodeSet := 1 + LengthOfNodeSet(NodeSet^.Next);
106 end;
107
108
109
110
111
112
113
114 function Distance(NodeSet : NodeType) : real;
115 {
116     NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
117     # of NodeSet for Distance >= 2
118     Function EuclideanDistance will find the Euclidean Distance
119         between the first node and the second in NodeSet.
120 }
121 var
122     Current : NodeType;
123 begin
124     Current := NodeSet;
125     if (LengthOfNodeSet(Current) <= 2)
126         then Distance := EuclideanDistance(Current)
127         else Distance := EuclideanDistance(Current)
128             + Distance(Current^.Next)
129 end;
130
131
132
133
134
135 procedure Copy(var NodeSet : NodeType; var MinSet : NodeType);
136 {
137     NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
138     This Procedure will duplicate the NodeSet in the other memory
139         storage.
140 }
141 var
142     NodeHolder, Current : NodeType;
143
144 begin
145     MinSet := nil;
146     NodeHolder := NodeSet;
147
148     new(MinSet);
149     MinSet^.Node := NodeSet^.Node;
150     MinSet^.Next := nil;

```



```

151      Current := MinSet;
152
153      while (NodeSet^.Next <> nil)
154      do begin
155          new(Current^.Next);
156          Current := Current^.Next;
157          NodeSet := NodeSet^.Next;
158          Current^.Node := NodeSet^.Node;
159          Current^.Next := nil;
160      end;
161      NodeSet := NodeHolder;
162  end;
163
164
165
166
167
168  procedure BranchAndBound(var NodeSet : NodeType;
169                          var MinSet : NodeType);
170  (
171      NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
172      This Procedure will find the the compensated nodes by
173                                          Branch and Bound Method.
174  )
175
176  var
177      N1, N2, N2holder,
178      Increament, MinDistance : real;
179      Head: NodeType;
180
181  begin
182      Increament := 0.01;
183      Head := NodeSet;
184      Copy(NodeSet,MinSet);
185
186      NodeSet := NodeSet^.Next;
187      N1 := NodeSet^.Node.z;
188      NodeSet := NodeSet^.Next;
189      N2 := NodeSet^.Node.z;
190      N2holder := N2;
191      NodeSet := Head;
192
193      MinDistance := Distance(NodeSet);
194
195      while (N1 > 0.0)
196      do begin
197          while (N2 > 0.0)
198          do begin
199              N2 := N2 - Increament;
200              Head := NodeSet;

```

```

201         NodeSet := NodeSet^.Next;
202         NodeSet := NodeSet^.Next;
203         NodeSet^.Node.z := N2;
204         NodeSet := Head;
205
206         if (MinDistance > Distance(NodeSet))
207         then begin
208             MinDistance := Distance(NodeSet);
209             Copy(NodeSet, MinSet);
210             end;
211         end;
212
213         N2 := N2holder;
214
215         N1 := N1 - Increament;
216         Head := NodeSet;
217         NodeSet := NodeSet^.Next;
218         NodeSet^.Node.z := N1;
219         NodeSet := Head;
220     end;
221 end;
222
223
224
225
226
227
228 procedure CreateNodes(var NodeSet : NodeType);
229 (
230     NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
231     This Procedure will create the NodeSet.
232 )
233 var
234     Current : NodeType;
235 begin
236     NodeSet := nil;
237     if not eof(BBinput)
238     then begin
239         new(NodeSet);
240         readln(BBinput, NodeSet^.Node.x,
241                 NodeSet^.Node.y,
242                 NodeSet^.Node.z);
243         NodeSet^.Next := nil;
244         Current := NodeSet;
245         while not eof(BBinput)
246         do begin
247             new(Current^.Next);
248             Current := Current^.Next;
249             readln(BBinput, Current^.Node.x,
250                     Current^.Node.y,

```

```

251
252
253
254
255
256
257
258
259
260
261
262 begin (____ MAIN ____ )
263     reset(BBinput);
264     rewrite(BBoutput);
265
266     CreateNodes(NodeSet);
267     writeln(BBoutput);
268     writeln(BBoutput);
269     writeln(BBoutput);
270     writeln(BBoutput, 'The original vertices by VGraph Algorithm');
271     writeln(BBoutput);
272     writeln(BBoutput, '_____ original distance = ',
273                                     Distance(NodeSet) :7:4);
274     PrintNodes(NodeSet);
275     BranchAndBound(NodeSet, MinSet);
276     writeln(BBoutput);
277     writeln(BBoutput);
278     writeln(BBoutput);
279     writeln(BBoutput, 'The vertices compensated by BranchAndBound');
280     writeln(BBoutput, '_____ ',
281                                     Distance(MinSet) :10:4);
282     PrintNodes(MinSet);
283     writeln(BBoutput);
284     writeln(BBoutput);
285     writeln(BBoutput, '
286     clock = ', clock);
287     system = ', sysclock);
288 end. (____ MAIN ____ )

```

Appendix F: I/O FILES for the Branch and Bound Algorithm

[BBinput]

```
3  2  4
7  4 10
8  8  9
4 11  2
```

[BBoutput]

The original vertices by VGraph Algorithm

<hr/>			original distance = 20.3283
3.0000	2.0000	4.0000	
7.0000	4.0000	10.0000	
8.0000	8.0000	9.0000	
4.0000	11.0000	2.0000	

The vertices compensated by BranchAndBound
13.7416

<hr/>		
3.0000	2.0000	4.0000
7.0000	4.0000	3.3400
8.0000	8.0000	2.7300
4.0000	11.0000	2.0000

```
clock  = 2301666
system = 10116
```

Appendix G: Simulation of the RCA

```

1  program RCAAlgorithm(RCAinput,RCAoutput);
2  {
3
4      Author   :   C. H. Chung
5
6      Version  :   2.0
7
8      Date     :   October 25, 1988
9
10
11     NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the graph.
12     This NodeSet is implemented by linked list, which node has
13     the record structure to represent the vertices.
14     Input file comes from RCAinput.
15     Output file is RCAoutput.
16     Determine Error to decide the accuracy.
17     Refer to Appendix C in RAL-TR-88-117.
18 }
19 type
20
21     PointType = record
22         x, y, z : real
23     end;
24
25     NodeType = ^Nodes;
26     Nodes = record
27         Node : PointType;
28         Next : NodeType
29     end;
30
31 var
32
33     RCAinput, RCAoutput : text;
34     NodeSet : NodeType;
35     Error : real;
36
37
38
39
40
41
42
43     procedure PrintNodes(NodeSet : NodeType);
44     {
45         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the graph.
46         This Procedure will print the NodeSet.
47     }
48 begin
49     if NodeSet = nil
50     then

```

```

51         else begin
52             writeln(RCAoutput, NodeSet^.Node.x :10:4,
53                                     NodeSet^.Node.y :10:4,
54                                     NodeSet^.Node.z :10:4);
55             writeln(RCAoutput);
56             PrintNodes(NodeSet^.Next)
57         end
58     end;
59
60
61
62
63
64
65     function EuclideanDistance(NodeSet : NodeType) : real;
66     {
67         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
68         This Function will calculate the Euclidean Distance
69         between the points in 3D.
70     }
71     var
72         Current : NodeType;
73         d1, d2, d3 : real;
74         x1, y1, z1,
75         x2, y2, z2 : real;
76     begin
77         Current := NodeSet;
78         x1 := Current^.Node.x;
79         y1 := Current^.Node.y;
80         z1 := Current^.Node.z;
81
82         Current := Current^.Next;
83         x2 := Current^.Node.x;
84         y2 := Current^.Node.y;
85         z2 := Current^.Node.z;
86
87         d1 := (x2 - x1) * (x2 - x1);
88         d2 := (y2 - y1) * (y2 - y1);
89         d3 := (z2 - z1) * (z2 - z1);
90
91         EuclideanDistance := sqrt(d1 + d2 + d3)
92     end;
93
94
95
96     procedure Reset(var NodeSet : NodeType);
97     {
98         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
99         Refer to Appendix C in RAL-TR-88-117.
100     }

```

```

101  var
102      Current : NodeType;
103      x0, y0, z0,
104      x1, y1, z1,
105      x2, y2, z2,
106      c1, c2, c3, c4,
107      p, q, r,
108      d1, d2, ed1, ed2 : real;
109  begin
110      Current := NodeSet;
111      x0 := Current^.Node.x;
112      y0 := Current^.Node.y;
113      z0 := Current^.Node.z;
114
115      Current := Current^.Next;
116      x1 := Current^.Node.x;
117      y1 := Current^.Node.y;
118      z1 := Current^.Node.z;
119
120      Current := Current^.Next;
121      x2 := Current^.Node.x;
122      y2 := Current^.Node.y;
123      z2 := Current^.Node.z;
124
125      c1 := (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0);
126      c2 := z2-z1;
127      c3 := (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1);
128      c4 := z1-z0;
129
130      p := c1-c3;
131      q := 2 *(c1*c2 + c3*c4);
132      r := c1*c2*c2 - c3*c4*c4;
133
134      if (p = 0)
135      then d1 := -r / q
136      else begin
137          d1 := (-q + sqrt(q*q - 4*p*r)) / (2*p);
138          d2 := (-q - sqrt(q*q - 4*p*r)) / (2*p);
139          ed1 := sqrt((x1-x0) * (x1-x0)
140                      + (y1-y0) * (y1-y0)
141                      + (z1-z0-d1) * (z1-z0-d1))
142                + sqrt((x2-x1) * (x2-x1)
143                      + (y2-y1) * (y2-y1)
144                      + (z2-z1+d1) * (z2-z1+d1));
145          ed2 := sqrt((x1-x0) * (x1-x0)
146                      + (y1-y0) * (y1-y0)
147                      + (z1-z0-d2) * (z1-z0-d2))
148                + sqrt((x2-x1) * (x2-x1)
149                      + (y2-y1) * (y2-y1)
150                      + (z2-z1+d2) * (z2-z1+d2));

```

```

151         if (ed1 > ed2)
152             then d1 := d2
153         end;
154
155         NodeSet^.Next^.Node.z := z1 - d1
156     end;
157
158
159
160
161
162     function LengthOfNodeSet(NodeSet : NodeType) : integer;
163     {
164         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
165         This Function will find the length of NodeSet.
166     }
167     begin
168         if (NodeSet^.Next = nil)
169             then LengthOfNodeSet := 1
170             else LengthOfNodeSet := 1 + LengthOfNodeSet(NodeSet^.Next)
171         end;
172
173
174
175
176
177
178     procedure Compensate(var NodeSet : NodeType);
179     {
180         NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
181         # of NodeSet for Compensate >= 3.
182         Procedure Reset will take the first 3 nodes in NodeSet
183         replace the second of the 3 nodes in NodeSet in
184         order to get the set of the compensated nodes
185     }
186     begin
187         if (LengthOfNodeSet(NodeSet) = 3)
188             then Reset(NodeSet)
189             else begin
190                 Reset(NodeSet);
191                 Compensate(NodeSet^.Next)
192             end
193     end;
194
195
196
197
198
199     function Distance(NodeSet : NodeType) : real;
200     {

```



```

201 NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
202 # of NodeSet for Distance >= 2
203 Function EuclideanDistance will find the Euclidean Distance
204 between the first node and the second in NodeSet.
205
206 var
207     Current : NodeType;
208 begin
209     Current := NodeSet;
210     if (LengthOfNodeSet(Current) <= 2)
211     then Distance := EuclideanDistance(Current)
212     else Distance := EuclideanDistance(Current)
213                   + Distance(Current^.Next)
214 end;
215
216
217
218
219
220
221 procedure RCA(var NodeSet : NodeType; Error : real);
222 {
223     NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.
224     Error = a permissible error
225     Function Distance will calculate the Euclidean Distance
226                                     through NodeSet.
227     Procedure Compensate will find the compensated nodes and
228                                     will return the set of these nodes.
229 }
230 var
231     Path1, Path2 : real;
232 begin
233     Path1 := Distance(NodeSet);
234     Compensate(NodeSet);
235     Path2 := Distance(NodeSet);
236     writeln(RCAoutput, '_____ compensated (Path1 - Path2) = ',
237             Path1 - Path2, ', ');
238     PrintNodes(NodeSet);
239     if (abs(Path1 - Path2) > Error)
240     then RCA(NodeSet, Error)
241 end;
242
243
244
245
246
247
248 procedure CreateNodes(var NodeSet : NodeType);
249 {
250     NodeSet = [S] U [N1,N2,N3 ...] U [G] searched by the VGraph.

```

```

251      This Procedure will create the NodeSet.
252      _____}
253  var
254      Current : NodeType;
255  begin
256      NodeSet := nil;
257      if not eof(RCAinput)
258      then begin
259          new(NodeSet);
260          readln(RCAinput, NodeSet^.Node.x,
261                NodeSet^.Node.y,
262                NodeSet^.Node.z);
263          NodeSet^.Next := nil;
264          Current := NodeSet;
265          while not eof(RCAinput)
266          do begin
267              new(Current^.Next);
268              Current := Current^.Next;
269              readln(RCAinput, Current^.Node.x,
270                    Current^.Node.y,
271                    Current^.Node.z);
272              Current^.Next := nil
273          end
274      end
275  end;
276
277
278
279
280
281
282  begin (_____ MAIN _____)
283      reset(RCAinput);
284      rewrite(RCAoutput);
285
286      Error := 0.00001;
287      CreateNodes(NodeSet);
288      writeln(RCAoutput);
289      writeln(RCAoutput);
290      writeln(RCAoutput);
291      writeln(RCAoutput, 'The original vertices by VGraph Algorithm');
292      writeln(RCAoutput);
293      writeln(RCAoutput, '_____ original distance = ',
294              Distance(NodeSet) :7:4);
295      PrintNodes(NodeSet);
296      RCA(NodeSet, Error);
297      writeln(RCAoutput);
298      writeln(RCAoutput);
299      writeln(RCAoutput);
300      writeln(RCAoutput, 'The vertices compensated by RCA');

```

```

301      writeln(RCAoutput, '_____');
302      PrintNodes(NodeSet);
303      writeln(RCAoutput);
304      writeln(RCAoutput);
305      writeln(RCAoutput, '      clock = ', clock);
306      writeln(RCAoutput, '      system = ', sysclock);
307 end. {_____ MAIN _____}

```

Appendix H: I/O FILES for the RCA

[RCAinput]

```

3  2  4
7  4 10
8  8  9
4 11  2

```

[RCAoutput]

The original vertices by VGraph Algorithm

<hr/>			original distance = 20.3283
3.0000	2.0000	4.0000	
7.0000	4.0000	10.0000	
8.0000	8.0000	9.0000	
4.0000	11.0000	2.0000	

<hr/>			compensated distance = 15.3916
3.0000	2.0000	4.0000	
7.0000	4.0000	6.6015	
8.0000	8.0000	4.5219	
4.0000	11.0000	2.0000	

<hr/>			compensated distance = 10.7529
3.0000	2.0000	4.0000	
7.0000	4.0000	4.2715	
8.0000	8.0000	3.2449	
4.0000	11.0000	2.0000	

<hr/>			compensated distance = 10.7529
3.0000	2.0000	4.0000	
7.0000	4.0000	3.6071	
8.0000	8.0000	2.8808	
4.0000	11.0000	2.0000	

3.0000	2.0000	4.0000	compensated distance = 13.7425
7.0000	4.0000	3.4177	
8.0000	8.0000	2.7770	
4.0000	11.0000	2.0000	

3.0000	2.0000	4.0000	compensated distance = 13.7416
7.0000	4.0000	3.3637	
8.0000	8.0000	2.7474	
4.0000	11.0000	2.0000	

3.0000	2.0000	4.0000	compensated distance = 13.7416
7.0000	4.0000	3.3482	
8.0000	8.0000	2.7389	
4.0000	11.0000	2.0000	

3.0000	2.0000	4.0000	compensated distance = 13.7416
7.0000	4.0000	3.3439	
8.0000	8.0000	2.7365	
4.0000	11.0000	2.0000	

The vertices compensated by RCA

3.0000	2.0000	4.0000
7.0000	4.0000	3.3439
8.0000	8.0000	2.7365
4.0000	11.0000	2.0000

clock	=	400
system	=	83

Appendix I: Simulation of the OPM

```
1 program OrthogonalProjectionMethod(OBJECT, PROJECTION);
2 {
3
4     Author   :   C. H. Chung
5
6     Version  :   2.3
7
8     Date     :   December 3, 1988
9
10
11
12     OPM;
13
14     .   To build the Grown Space Obstacles in 3D by OPM
15     .   INPUT  FILE : OBJECT
16     .   OUTPUT FILE : PROJECTION
17
18
19     This program will build the Grown Space Obstacles in 3D.
20 }
21 type
22     Point2D = record
23         x, y : real;
24     end;
25     Vertice2D = ^Node2D;
26     Node2D = record
27         Node : Point2D;
28         Next : Vertice2D
29     end;
30
31     Point3D = record
32         x, y, z : real;
33     end;
34     Vertice3D = ^Node3D;
35     Node3D = record
36         Node : Point3D;
37         Next : Vertice3D
38     end;
39 var
40     OBJECT, PROJECTION : text;
41     LinkedVertices : Vertice3D;
42
43
44
45
46
47     procedure OPM(var LinkedVertices : Vertice3D);
48     {
49
50         Author   :   C. H. Chung
```

```

51
52      Version : 2.3
53
54      Date : December 3, 1988
55
56      _____
57
58      OPM(LinkedVertices);
59
60      . To build the Grown Space Obstacles in 3D by OPM.
61      . INPUT FILE : OBJECT
62      . OUTPUT FILE : PROJECTION
63
64
65      This program will build the Grown Space Obstacles.
66
67      hh : the horizontal length of the object
68
69      vv : the vertical length of the object
70
71      rr : the sliced angle for rotational Grown Space Obstacles.
72      _____}
73
74      const
75      Pi = 3.141592;      {Radian}
76
77      var
78      Object : Vertice3D;
79      ObjectXY, ObjectYZ, ObjectXZ : Vertice2D;
80      GrownXY, GrownYZ, GrownXZ : Vertice2D;
81      rr : real;
82      hh1, hh2, hh3,
83      vv1, vv2, vv3 : real;
84
85
86
87
88
89      procedure Print2Dvertice(List : Vertice2D);
90      {
91      This procedure will print the Linked List of the
92      shortest path.
93      _____}
94
95      begin
96      if List = nil
97      then writeln(PROJECTION)
98      else begin
99          writeln(PROJECTION, List^.Node.x :10:4,
100                List^.Node.y :10:4);
101          Print2Dvertice(List^.Next)

```



```

101          end
102      end;
103
104
105
106
107
108
109
110  procedure Print3Dvertice(List : Vertice3D);
111  (
112      This procedure will print the Linked List of the
113      shortest path.
114  )
115  begin
116      if List = nil
117      then writeln(PROJECTION)
118      else begin
119          writeln(PROJECTION, List^.Node.x :10:4,
120                  List^.Node.y :10:4,
121                  List^.Node.z :10:4,
122                  Print3Dvertice(List^.Next)
123          end
124      end;
125
126
127
128
129
130
131  procedure CreateObject(var Object : Vertice3D);
132  (
133      This procedure creates the object from the input file
134      by the linked list.
135  )
136  var
137      Current : Vertice3D;
138  begin
139      Object := nil;
140      if not eof(OBJECT)
141      then begin
142          new(Object);
143          readln(OBJECT, Object^.Node.x,
144                  Object^.Node.y,
145                  Object^.Node.z);
146          Object^.Next := nil;
147          Current := Object;
148          while not eof(OBJECT)
149          do begin
150              new(Current^.Next);

```

```

151         Current := Current^.Next;
152         readln(OBJECT, Current^.Node.x,
153               Current^.Node.y,
154               Current^.Node.z);
155         Current^.Next := nil
156     end
157 end
158 end;
159
160
161
162
163
164
165
166 procedure GrownObject(var Object, Grown : Vertice2D;
167                      hh, vv, rr : real);
168 {
169     This procedure builds the Grown Space Obstacles.
170
171     where h : horizontal
172           v : vertical
173
174     0 < q < Pi/2
175
176         a1 = (A1x,A1y) + h(-cos(q),-sin(q))
177         a2 = (A1x,A1y)
178         a3 = (A2x,A2y)
179         a4 = (A2x,A2y) + v(sin(q),-cos(q))
180         a5 = (A3x,A3y) + v(sin(q),-cos(q))
181         a6 = (a5x,a5y) + h(-cos(q),-sin(q))
182         a8 = (A4x,A4y) + h(-cos(q),-sin(q))
183         a7 = (a8x,a8y) + v(sin(q),-cos(q))
184
185     Pi/2 < q < Pi
186
187         q = q - Pi/2
188         temp = h      (to swap h and v)
189         h = v
190         v = temp
191
192     q = 0
193         Delete a2, a4, a6, a8.
194
195     q = Pi/2
196         Swap h and v.
197         Delete a2, a4, a6, a8.
198 }
199 var
200     Current, Head : Vertice2D;

```

```

201 begin
202     Current := nil;
203     new(Current);
204     Current^.Node.x := Object^.Node.x - hh * cos(rr);
205     Current^.Node.y := Object^.Node.y - hh * sin(rr);
206     Current^.Next := nil;
207
208     Head := Object;
209     Grown := Current;
210
211     new(Current^.Next);
212     Current := Current^.Next;
213     Current^.Node.x := Object^.Node.x;
214     Current^.Node.y := Object^.Node.y;
215     Current^.Next := nil;
216
217     Object := Object^.Next;
218     new(Current^.Next);
219     Current := Current^.Next;
220     Current^.Node.x := Object^.Node.x;
221     Current^.Node.y := Object^.Node.y;
222     Current^.Next := nil;
223
224     new(Current^.Next);
225     Current := Current^.Next;
226     Current^.Node.x := Object^.Node.x + vv * sin(rr);
227     Current^.Node.y := Object^.Node.y - vv * cos(rr);
228     Current^.Next := nil;
229
230     Object := Object^.Next;
231     new(Current^.Next);
232     Current := Current^.Next;
233     Current^.Node.x := Object^.Node.x + vv * sin(rr);
234     Current^.Node.y := Object^.Node.y - vv * cos(rr);
235     Current^.Next := nil;
236
237     new(Current^.Next);
238     Current := Current^.Next;
239     Current^.Node.x := Object^.Node.x + vv*sin(rr) - hh*cos(rr);
240     Current^.Node.y := Object^.Node.y - vv*cos(rr) - hh*sin(rr);
241     Current^.Next := nil;
242
243     Object := Object^.Next;
244     new(Current^.Next);
245     Current := Current^.Next;
246     Current^.Node.x := Object^.Node.x + vv*sin(rr) - hh*cos(rr);
247     Current^.Node.y := Object^.Node.y - vv*cos(rr) - hh*sin(rr);
248     Current^.Next := nil;
249
250     new(Current^.Next);

```

```

251     Current := Current^.Next;
252     Current^.Node.x := Object^.Node.x - hh * cos(rr);
253     Current^.Node.y := Object^.Node.y - hh * sin(rr);
254     Current^.Next := nil;
255
256     Object := Head;
257
258     if (rr = 0) or (abs(rr - Pi/2) < 0.001)
259     then begin
260         Current := nil;
261         new(Current);
262         Current^.Node := Grown^.Node;
263         Current^.Next := nil;
264
265         Head := nil;
266         Head := Current;
267         while Grown^.Next^.Next <> nil
268         do begin
269             new(Current^.Next);
270             Current := Current^.Next;
271             Grown := Grown^.Next^.Next;
272             Current^.Node := Grown^.Node;
273             Current^.Next := nil;
274         end;
275         Grown := Head;
276     end;
277 end;
278
279
280
281
282
283
284 procedure OrthogonalProjection(Object : Vertice3D;
285                               var ObjectXY, ObjectYZ, ObjectXZ : Vertice2D);
286 {
287     This procedure project Object in 3D into 3 Objects in 2D.
288 }
289 var
290     Current : Vertice3D;
291     CurrentXY,
292     CurrentYZ,
293     CurrentXZ : Vertice2D;
294     X1, X2,
295     Y1, Y2,
296     Z1, Z2 : real;
297
298 begin
299
300     Current := Object;

```

```

301
302     ObjectXY := nil;
303     ObjectYZ := nil;
304     ObjectXZ := nil;
305
306     X1 := Current^.Node.x;
307     Y1 := Current^.Node.y;
308     Z1 := Current^.Node.z;
309
310     Current := Current^.Next;
311     Y2 := Current^.Node.y;
312
313     Current := Current^.Next;
314     X2 := Current^.Node.x;
315
316     Current := Current^.Next;
317     Current := Current^.Next;
318     Z2 := Current^.Node.z;
319
320     new(ObjectXY);
321     ObjectXY^.Node.x := X1;
322     ObjectXY^.Node.y := Y2;
323     ObjectXY^.Next := nil;
324
325     new(ObjectYZ);
326     ObjectYZ^.Node.x := Y1;
327     ObjectYZ^.Node.y := Z2;
328     ObjectYZ^.Next := nil;
329
330     new(ObjectXZ);
331     ObjectXZ^.Node.x := X1;
332     ObjectXZ^.Node.y := Z2;
333     ObjectXZ^.Next := nil;
334
335     CurrentXY := ObjectXY;
336     CurrentYZ := ObjectYZ;
337     CurrentXZ := ObjectXZ;
338
339     new(CurrentXY^.Next);
340     CurrentXY := CurrentXY^.Next;
341     CurrentXY^.Node.x := X2;
342     CurrentXY^.Node.y := Y2;
343     CurrentXY^.Next := nil;
344
345     new(CurrentYZ^.Next);
346     CurrentYZ := CurrentYZ^.Next;
347     CurrentYZ^.Node.x := Y2;
348     CurrentYZ^.Node.y := Z2;
349     CurrentYZ^.Next := nil;
350

```

```

351     new(CurrentXZ^.Next);
352     CurrentXZ := CurrentXZ^.Next;
353     CurrentXZ^.Node.x := X2;
354     CurrentXZ^.Node.y := Z2;
355     CurrentXZ^.Next := nil;
356
357     new(CurrentXY^.Next);
358     CurrentXY := CurrentXY^.Next;
359     CurrentXY^.Node.x := X2;
360     CurrentXY^.Node.y := Y1;
361     CurrentXY^.Next := nil;
362
363     new(CurrentYZ^.Next);
364     CurrentYZ := CurrentYZ^.Next;
365     CurrentYZ^.Node.x := Y2;
366     CurrentYZ^.Node.y := Z1;
367     CurrentYZ^.Next := nil;
368
369     new(CurrentXZ^.Next);
370     CurrentXZ := CurrentXZ^.Next;
371     CurrentXZ^.Node.x := X2;
372     CurrentXZ^.Node.y := Z1;
373     CurrentXZ^.Next := nil;
374
375     new(CurrentXY^.Next);
376     CurrentXY := CurrentXY^.Next;
377     CurrentXY^.Node.x := X1;
378     CurrentXY^.Node.y := Y1;
379     CurrentXY^.Next := nil;
380
381     new(CurrentYZ^.Next);
382     CurrentYZ := CurrentYZ^.Next;
383     CurrentYZ^.Node.x := Y1;
384     CurrentYZ^.Node.y := Z1;
385     CurrentYZ^.Next := nil;
386
387     new(CurrentXZ^.Next);
388     CurrentXZ := CurrentXZ^.Next;
389     CurrentXZ^.Node.x := X1;
390     CurrentXZ^.Node.y := Z1;
391     CurrentXZ^.Next := nil;
392 end;
393
394
395
396
397
398
399 procedure ReconstructObject(ObjectXY, ObjectYZ, ObjectXZ :
400                               Vertice2D; var LinkedVertices : Vertice3D);

```

```

401
402
403      This procedure will reconstruct the object from the
404      3 projected images in 2D.
405
406  var
407      Current : Vertice3D;
408      CurrentXY,
409      CurrentYZ,
410      CurrentXZ : Vertice2D;
411      X1, Y1, Z1,
412      X2, Y2, Z2 : real;
413  begin
414      LinkedVertices := nil;
415      new(LinkedVertices);
416      Current := LinkedVertices;
417      LinkedVertices^.Next := nil;
418
419      CurrentXY := ObjectXY;
420      CurrentYZ := ObjectYZ;
421      CurrentXZ := ObjectXZ;
422
423      while CurrentXY <> nil do
424          begin
425              X1 := CurrentXY^.Node.x;
426              Y1 := CurrentXY^.Node.y;
427              CurrentXY := CurrentXY^.Next;
428              while CurrentYZ <> nil do
429                  begin
430                      Y2 := CurrentYZ^.Node.x;
431                      Z1 := CurrentYZ^.Node.y;
432                      CurrentYZ := CurrentYZ^.Next;
433                      if (Y1 = Y2)
434                          then while CurrentXZ <> nil do
435                              begin
436                                  X2 := CurrentXZ^.Node.x;
437                                  Z2 := CurrentXZ^.Node.y;
438                                  CurrentXZ := CurrentXZ^.Next;
439                                  if (Z1 = Z2) and (X1 = X2)
440                                      then begin
441                                          Current^.Node.x := X1;
442                                          Current^.Node.y := Y1;
443                                          Current^.Node.z := Z1;
444                                          Current^.Next := nil;
445                                          if (CurrentXY = nil) and
446                                              (CurrentYZ = nil) and
447                                              (CurrentXZ = nil)
448                                              then
449                                                  else begin
450

```

```

451                                     Current := Current^.Next;
452                                     end;
453                                     end;
454                                     end;
455                                     CurrentXZ := ObjectXZ;
456                                     end;
457                                     CurrentYZ := ObjectYZ;
458                                     end;
459                                     Current := LinkedVertices;
460                                     while Current^.Next^.Next <> nil
461                                     do Current := Current^.Next;
462                                     Current := nil;
463                                     end;
464
465
466
467     begin {_____ OPM _____}
468         reset(OBJECT);
469         rewrite(PROJECTION);
470
471         rr := Pi/6;
472         rr := 0;
473         hh1 := 1.0;
474         hh2 := 1.5;
475         hh3 := 1.0;
476         vv1 := 1.5;
477         vv2 := 0.5;
478         vv3 := 0.5;
479
480         CreateObject(Object);
481         writeln(PROJECTION, ' Object in 3D:');
482         writeln(PROJECTION);
483         Print3Dvertice(Object);
484
485         OrthogonalProjection(Object, ObjectXY, ObjectYZ, ObjectXZ);
486         writeln(PROJECTION);
487         writeln(PROJECTION);
488         writeln(PROJECTION, ' Image projected in 2D:');
489         writeln(PROJECTION);
490         writeln(PROJECTION, '      (X,Y) projection');
491         Print2Dvertice(ObjectXY);
492         writeln(PROJECTION);
493         writeln(PROJECTION, '      (Y,Z) projection');
494         Print2Dvertice(ObjectYZ);
495         writeln(PROJECTION);
496         writeln(PROJECTION, '      (X,Z) projection');
497         Print2Dvertice(ObjectXZ);
498
499         GrownObject(ObjectXY, GrownXY, hh1, vv1, rr);
500         GrownObject(ObjectYZ, GrownYZ, hh2, vv2, rr);

```



```

501      GrownObject(ObjectXZ, GrownXZ, hh3, vv3, rz);
502
503      writeln(PROJECTION);
504      writeln(PROJECTION);
505      writeln(PROJECTION, ' Grown Image projected in 2D:');
506      writeln(PROJECTION);
507      writeln(PROJECTION);
508      writeln(PROJECTION, '      (X,Y) Grown Image');
509      Print2Dvertice(GrownXY);
510      writeln(PROJECTION);
511      writeln(PROJECTION, '      (Y,Z) Grown Image');
512      Print2Dvertice(GrownYZ);
513      writeln(PROJECTION);
514      writeln(PROJECTION, '      (X,Z) Grown Image');
515      Print2Dvertice(GrownXZ);
516
517      ReconstructObject(GrownXY, GrownYZ, GrownXZ, LinkedVertices);
518      writeln(PROJECTION);
519      writeln(PROJECTION);
520      writeln(PROJECTION, ' Object in 3D reconstructed by OPM');
521      writeln(PROJECTION);
522      Print3Dvertice(LinkedVertices);
523
524      end; ( of Procedure OPM )
525
526
527
528
529 begin (___ Main ___)
530     OPM(LinkedVertices);
531 end. (___ of Main ___)

```

Appendix J: I/O FILES for the OPM

[OBJECT]

7	5	3
7	10	3
14	10	3
14	5	3
7	5	12
7	10	12
14	10	12
14	5	12

[PROJECTION]

Object in 3D:

7.0000	5.0000	3.0000
7.0000	10.0000	3.0000
14.0000	10.0000	3.0000
14.0000	5.0000	3.0000
7.0000	5.0000	12.0000
7.0000	10.0000	12.0000
14.0000	10.0000	12.0000
14.0000	5.0000	12.0000

Image projected in 2D:

(X,Y) projection

7.0000	10.0000
14.0000	10.0000
14.0000	5.0000
7.0000	5.0000

(Y,Z) projection

5.0000	12.0000
10.0000	12.0000
10.0000	3.0000
5.0000	3.0000

(X,Z) projection	
7.0000	12.0000
14.0000	12.0000
14.0000	3.0000
7.0000	3.0000

Grown Image projected in 2D:

(X,Y) Grown Image	
6.0000	10.0000
14.0000	10.0000
14.0000	3.5000
6.0000	3.5000

(Y,Z) Grown Image	
3.5000	12.0000
10.0000	12.0000
10.0000	2.5000
3.5000	2.5000

(X,Z) Grown Image	
6.0000	12.0000
14.0000	12.0000
14.0000	2.5000
6.0000	2.5000

Object in 3D reconstructed by OPM:

6.0000	10.0000	12.0000
6.0000	10.0000	2.5000
14.0000	10.0000	12.0000
14.0000	10.0000	2.5000
14.0000	3.5000	12.0000
14.0000	3.5000	2.5000
6.0000	3.5000	12.0000
6.0000	3.5000	2.5000

